Practical framework for contract-based concurrent object-oriented programming

Piotr Nienaltowski

Doctoral Thesis ETH No. 17061

Practical framework for contract-based concurrent object-oriented programming

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZURICH)

for the degree of
Doctor of Sciences (Dr. sc. ETH Zurich)

presented by
Piotr Nienaltowski
DEA Université Joseph Fourier/INPG Grenoble, France
born on 30 June 1976
Polish citizen

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, ETH Zurich, examiner
Prof. Dr. Peter Müller, ETH Zurich, co-examiner
Prof. Dr. Jonathan Ostroff, York University, Toronto, co-examiner

2007

Foreword

CONCURRENT programming, we have been told many times, is difficult. (The adjective *difficult* is usually embellished in this context with one of: *inherently*, *intrinsically*, *extremely*, or at least *very*.) Boyapati, Lee, and Rinard [28] warn against the pitfalls of widely used multithreading:

> Multithreaded programming is difficult and error-prone. Synchronization errors (...) are among the most difficult programming errors to detect, reproduce, and eliminate.

Sutter and Larus [135] point the mismatch between existing techniques and practical needs:

> Not only are today's languages and tools inadequate to transform applications into parallel programs, but also it is difficult to find parallelism in mainstream applications, and — worst of all — concurrency requires programmers to think in a way humans find difficult.

Harris, Herlihy et al. [63] observe the lack of modularity in current programming models:

> A particular source of concern is that even correctly implemented concurrency abstractions cannot be composed together to form larger abstractions.

The research effort described here stems from an optimistic belief that, although orchestrating several concurrent activities is more complicated than performing a single one, well-established software engineering techniques can help reduce the complexity of this task. Simplicity, abstraction, and modularity brought by object technology have been shown to improve dramatically the quality of sequential software; I have decided to learn how to apply them to solving concurrency problems. This dissertation is a report from my research journey into the fascinating world of object-oriented programming and Design by Contract to discover simple and convenient techniques for building high-quality concurrent systems.

Abstract

CONCURRENCY, in its many variants from multithreading to multiprocessing, distributed computing, Internet applications, and Web services, has become a required component of ever more types of systems, including some that are traditionally thought of as sequential. The software industry badly needs a concurrent programming technique enjoying the same simplicity and inspiring the same confidence as the accepted constructs of sequential programming. The object-oriented framework presented in this thesis — SCOOP — provides a suitable support to solving a large class of concurrency problems. It carries the advantages of object technology and Design by Contract to the concurrent context: concurrent software can be understood, analysed, written, and reused in a much simpler manner than with other state-of-the-art techniques.

We start from a previous version of the model: SCOOP_97. It is an interesting candidate for modelling concurrent applications because it takes advantage of the existing synergies between object-oriented concepts and concurrency. At the outset of this work, no implementation of SCOOP_97 was available and the conceptual implications of the model had not been fully worked out. Also missing was a detailed comparison with other approaches. This thesis fills these gaps by carrying out an in-depth analysis of the model, identifying inconsistencies, proposing adequate solutions to the encountered problems, extending the model, formalising it, and providing an implementation. The main results are:

- An enriched type system to detect and eliminate potential atomicity violations.
- A generalised semantics of contracts, applicable in concurrent and sequential contexts.
- A flexible locking policy to optimise the use of resources and to minimise the danger of deadlocks.
- A seamless integration of the concurrency model with Eiffel: a full-blown object-oriented language.
- A library implementation of SCOOP and a compiler which type-checks SCOOP code and translates it into pure Eiffel with embedded library calls; a supporting library of advanced concurrency mechanisms is also provided.

Our framework supports the essential object-oriented mechanisms: (multiple) inheritance, polymorphism, dynamic binding, genericity, expanded and attached (non-nullable) types, and agents. It is implemented as an extension of Eiffel but the results of this work are directly applicable to other object-oriented languages that support Design by Contract, e.g. Spec♯ and JML/Java.

Kurzfassung

NEBENLÄUFIGKEIT, in seinen vielen Varianten, angefangen bei Multithreading und Multiprocessing über Verteilte Systeme und Internet-Anwendungen bis hin zu Web-Services, ist ein notwendiger Bestandteil von immer mehr Systemen, auch solchen, welche früher als klassisch sequenziell galten. Die Softwareindustrie benötigt dringend Techniken zur Erstellung von nebenläufigen Programmen, welche so einfach und zuverlässig sind wie die verbreiteten Ansätze für sequenzielle Programmierung. Das objektorientierte Modell und Framework SCOOP, welches Thema dieser Arbeit ist, unterstützt in angemessener Weise die Lösung eines breiten Feldes an Problemstellungen. Es überträgt die Vorteile von Objekttechnologie und Design by Contract auf das Gebiet der Nebenläufigkeit: nebenläufige Anwendungen werden viel einfacher verstanden, analysiert, implementiert und wiederverwendet.

Ausgegangen sind wir von einer bestehenden Version des Modells, welche ein interessanter Kandidat für die Modellierung von nebenläufigen Anwendungen ist, weil sie die vorhandenen Beziehungen zwischen objektorientierten Konzepten und Nebenläufigkeit ausnutzt. Zu Beginn war weder eine Implementierung vorhanden, noch waren die konzeptionellen Auswirkungen des Modells vollständig ausgearbeitet. Auch fehlte ein detaillierter Vergleich mit anderen Ansätzen. Diese Arbeit schliesst diese Lücken durch eine tiefgreifende Analyse des Modells. Inkonsistenzen werden identifiziert und adäquate Lösungen werden vorgeschlagen. Das Modell wird formalisiert und eine Implementierung bereitgestellt. Die wesentlichen Ergebnisse sind:

- Ein angereichertes Typensystem, welches mögliche Probleme mit der Atomarität aufdeckt.
- Ein verallgemeinertes Verständnis von Contracts, welches in nebenläufigen und sequenziellen Anwendungen gleichermassen gilt.
- Ein flexibles Verfahren des Locking, welches Ressourcen optimal ausnutzt.
- Eine nahtlose Integration des Modells in eine vollständige, objektorientierte Programmiersprache.
- Eine Implementierung von SCOOP als Bibliothek und ein typüberprüfender Compiler, welcher den Code in reines Eiffel unter Verwendung der Bibliothek übersetzt; eine weitere Hilfsbibliothek mit fortgeschrittenen Mechanismen für Nebenläufigkeit wird ausserdem bereitgestellt.

Unser Framework unterstützt die wesentlichen objektorientierten Mechanismen: Vererbung, Polymorphie, Generizität, sowie Agents. Es ist als Erweiterung der Programmiersprache Eiffel umgesetzt, aber die Resultate sind direkt auf andere objektorientierte Sprachen mit Design by Contract anwendbar, beispielsweise Spec♯ oder JML/Java.

Acknowledgments

Many persons contributed to the completion of this work. First of all, I would like to thank my advisor Bertrand Meyer for giving me the opportunity to work with him and discover the beauty of object technology. His informative criticism has guided my research efforts, bringing this work to sharper focus. I have particularly appreciated his insistence on excellence and practicality of the undertaken research, and the constant encouragement to develop various professional and personal skills.

I am grateful to Peter Müller and Jonathan Ostroff for acting as reviewers of this dissertation. Their perspicacious comments have helped improve this work. Peter's course on semantics of programming languages and his research on ownership types provided much of the impetus for this work. I have benefited greatly from numerous discussions with Jonathan and his students.

Jean-Raymond Abrial's teaching talent and unlimited enthusiasm for proofs have helped overcome my initial fears of formal methods. I have appreciated his cheerful attitude which has proved infectious in many lectures and meetings of the ETH Formal Methods Club.

I am indebted to Richard Paige, Phil Brooke, Haifeng Huang, Faraz Torshizi, and Jeremy Jacob for the hot debates on SCOOP which have contributed to the better understanding of the model.

The friendly and motivating environment at the Chair of Software Engineering has made my stay at ETH an enjoyable and fruitful experience. My warmest thanks go to all my present and former colleagues: Karine Arnout, Volkan Arslan, Arnaud Bailly, Stephanie Balzer, Till Bay, Ruth Bürkli, Susanne Cech, Ilinca Ciupa, Ádám Darvas, Werner Dietl, Vijay D'silva, Patrick Eugster, Claudia Günthart, Stefan Hallerstede, Hermann Lehner, Andreas Leitner, Lisa Liu, Luc de Louw, Farhad Mehta, Martin Nordio, Manuel Oriol, Michela Pedroni, Marco Piccioni, Joseph Ruskiewicz, Bernd Schoeller, Sébastien Vaucouleur, Laurent Voisin, and Jenny Xiaohui. It has been a great pleasure to work on SCOOP-related projects with a number of smart students: Andreas Compeer, Daniel Moser, Yann Müller, Christopher Nenning, Gabriel Petrovay, and Ganesh Ramanathan. I am particularly thankful to all the students who took the Concurrent Programming II course in 2006. Their feedback was invaluable; so was their patience with the early versions of SCOOP tools.

I would like to thank my family for their continuous encouragement and belief in the success of my projects and endeavours.

Finally, the very special thanks go to Marie-Hélène Ng Cheong Vee for her support, understanding, and amazing patience. Thank you for putting up with me in the tough times preceding the completion of this work. I promise some improvement in the future.♡

Contents

1 Introduction

THE real world is concurrent — several things happen at the same time. Computer systems that are intended to model the world must account for its concurrent nature. Concurrent programming has become a required component of ever more types of systems, including some that were traditionally thought of as sequential. There are numerous examples of such applications: banking systems with multiple ATMs, avionics systems, booking systems, multi-user operating systems. Very often, sequential tasks may be parallelised and solved faster using several computing units working concurrently. Reactive systems which interact with their environment can be naturally modelled as concurrent systems [35]. When many activities are performed concurrently, they need to communicate and coordinate. Coordination should ensure that there is no harmful interference which may falsify the results of computation. Also, no activity should be infinitely prevented from progressing by others.

The industry is still looking for a good way to produce concurrent applications. The contrast with sequential programming is stark: there, a widely accepted set of ideas — standard control and data structuring techniques, modularity and information hiding, object-oriented principles — have displaced the lower-level concepts that used to predominate [94]. But the techniques commonly used to produce concurrent applications are still elementary and often haphazard. The explicit specification and control of low-level parallelism in multithreading models, e.g. in Java [79] and C♯ [46], is a source of new programming errors due to incorrect synchronisation. Three main types of synchronisation errors can be identified:

- *Data race*: a situation where different threads simultaneously access the same data without ordering, with at least one thread modifying the data. Data races lead to data inconsistency and unintended non-determinism.
- *Atomicity violation*: unwanted interleaving of operations performed by different threads, leading to harmful interference between threads which results in an inconsistent state of the system.
- *Deadlock*: a situation where synchronisation between threads causes a cyclic wait, e.g. thread A is waiting for thread B, B is waiting for C, and C is waiting for A, which prevents the involved threads from progressing.

A suitable concurrency model is needed that provides the basic safety and liveness guarantees, shielding programmers from these common mistakes and errors. Furthermore, the model should be flexible enough to provide support for many kinds of concurrent systems.

Is there a simpler and better basis for concurrent programming: a *disciplined* approach to building high-quality software systems? No unique solution will cater for the needs of all

possible applications of concurrency but is it possible to devise a simple technique, applicable to a large number of problems, that helps programmers design and develop correct and reusable concurrent programs in a modular way with little more effort than sequential ones?

Benefits of object technology

Object-oriented software construction is the building of software systems as collections of classes corresponding to well-defined data abstractions. These collections are structured using two inter-class relations: client and inheritance [94]. At run time, a system is represented by a set of objects (instances of classes) which communicate through feature calls. Object technology and its underlying principles permit building software that satisfies a number of quality factors. Of particular interest are:

- *Modularity*: the possibility to build a software system from smaller components, and to reason about its properties based on the properties of these components.
- *Ease of reuse*: the applicability of existing software elements to different contexts, and their utility as building blocks for new software systems.
- *Extendibility*: the possibility to easily change and extend existing components and systems.

Modularity and ease of reuse can be improved by minimising the coupling between software modules, and maximising the cohesion of single modules. As observed by Meyer [94], two techniques are essential for improving extendibility:

- *Design simplicity*: a simple architecture will always be easier to adapt to changes than a complex one.
- *Decentralisation*: the more autonomous the modules, the higher the likelihood that a simple change will affect just one module, or a small number of modules, rather than triggering off a chain reaction of changes over the whole system.

When properly applied, object technology permits to improve all these quality factors in sequential software, thanks to the use of the following principles and mechanisms:

- *Abstraction and information hiding*
  Abstraction permits stripping away irrelevant details, e.g. how a given feature is implemented. An abstract concept may have many different implementations; clients should be able to use a class through a clearly defined interface, without the need to see the implementation details. Information hiding is an important means to achieve high levels of abstraction. It is possible for the author of a class to decide that a feature is available to all clients, to no client, or to specified clients only. It is also possible to write a class as *deferred*, i.e. specified but not fully implemented. Deferred classes are useful for analysis and design because they allow capturing the essential aspects of a system while leaving details to a later stage.

- *Inheritance*
  Software development usually involves a large number of classes; many are variants of others. A class may inherit from other classes, thus incorporating the others' features in addition to its own. Inheritance is a convenient tool for abstraction and specialisation; it helps explore relations between the concepts modelled by classes, and it results in a clear and compact software structure.

- *Static typing*
  A well-defined type system guarantees run-time type safety, i.e. the absence of run-time errors of certain kinds, by enforcing a number of type declarations and consistency rules. Static typing increases the confidence in the correctness of a system before the system is executed.

- *Polymorphism and dynamic binding*
  Entities may be attached to objects of various types. In a typed language, such polymorphism is not arbitrary: it is controlled by the type rules. Calling a feature on an entity triggers the feature corresponding to the actual run-time type of the corresponding object, which may vary in different executions of the call. The run-time choice of the adequate version of a feature is referred to as *dynamic binding*.

Carrying the advantages of object technology to a concurrent context seems to be a natural next step in the development of software engineering techniques. Unfortunately, previous attempts at bringing together concurrency and object-oriented techniques have not been very successful. Existing concurrent object-oriented languages (see chapter 3) provide a limited support for advanced techniques such as inheritance, polymorphism, and genericity, and they fail to take full advantage of modularity, extendibility, and the potential for reuse offered by object technology. Löhr [86] observes that it is important for practical software engineering that both sequential and concurrent programs can be developed within the same framework, and the boundary between sequentiality and concurrency be no impediment to reuse. This calls for a full integration of concurrency and object-oriented techniques, to minimise the syntactic and semantic differences of sequential and concurrent code.

Can the full power of object-oriented techniques be unleashed in a concurrent context? Can concurrent software be designed, analysed, and implemented in a modular manner, and reused in a simple and efficient way?

Benefits of Design by Contract

Design by Contract [92, 94, 99] permits equipping class interfaces with contracts which express the mutual obligations of clients and suppliers. There are three main categories of assertions: *preconditions*, *postconditions*, and *invariants*. Routine preconditions specify the obligations on the client and the guarantee given to the supplier (routine implementor). Conversely, routine postconditions express the obligation on the supplier and the guarantee given to the client. Class invariants express the correctness criteria of a given class: an instance of a class is consistent only if the corresponding invariant holds in every observable state. Additionally, *loop assertions* facilitate proofs of loop correctness and termination. The modular design fostered by encapsulation and Design by Contract reduces the complexity of software: correctness considerations can be confined to the boundaries of components (classes) which can be proved and tested in

isolation. Clients can rely on the interface of a class without the need to know its implementation details. Through a precise definition of every module's claims and responsibilities, a significant degree of trust in large software systems can be attained.

Design by Contract has been applied successfully to sequential programming. It has proved useful not only as a method of analysis and design but also during implementation and testing. As pointed out in [94], writing the assertions at the same time as writing the software — or indeed before writing the software — brings tremendous benefits:

- Producing software that is correct from the start because it is designed to be correct.
- Getting a much better understanding of the problem and its eventual solutions.
- Facilitating the task of software documentation.
- Providing a systematic basis for testing and debugging.

This results in improved modularity, reusability, and extendibility of sequential software. Can Design by Contract bring all these improvements to concurrent software as well?

1.1 Thesis statement

This dissertation establishes the following theses:

- Object-oriented techniques coupled with Design by Contract constitute an adequate basis for the modular development of correct concurrent programs.
- Sequentiality is a particular case of concurrency; consequently, the semantics of object-oriented mechanisms — feature call, feature application, argument passing, and contracts — can be generalised to cover both the concurrent and the sequential contexts. There is no clash between concurrency and such essential object-oriented techniques as inheritance, polymorphism, dynamic binding, genericity, and agents; all of them can be fully supported.
- Important information relative to concurrent execution based on asynchronous calls can be expressed in types; a simple type system can eliminate potential atomicity violations in the presence of asynchrony without restricting the usability of the programming language.
- The proposed approach is applicable to full-fledged object-oriented languages with inheritance, polymorphism, and genericity; it can be implemented in practice.

1.2 Goals and evaluation criteria

My main goal is to bring concurrent programming to the same level of abstraction and convenience as sequential programming. This applies to several activities associated with software construction: design, implementation, analysis, verification, and reuse. Object-oriented techniques and Design by Contract, which constitute the basis of the Eiffel approach [92, 94], have been very successful at improving the quality of sequential software. This study is an attempt

to carry their advantages to the concurrent context. The intended practical outcome of this work is SCOOP: an O-O contract-based framework for concurrent programming.

I want to demonstrate that there is no intrinsic conflict between object-oriented concepts and concurrent programming. On the contrary: the full power of object technology can only be unleashed in a concurrent context. This claim is against the usual view of concurrency and object technology as two different, largely incompatible worlds. Although existing concurrent object-oriented programming languages (see chapter 3) provide only a limited support for object-oriented mechanisms, I do not see why it is necessary to sacrifice the extent to which O-O concepts, in particular inheritance, are supported. The apparent problems are caused by insufficient understanding of object-oriented mechanisms; therefore, studying the relationship between concurrent and O-O abstractions and capturing their intended semantics are important goals of my research. I try to take advantage of the implicit concurrency present in O-O mechanisms, e.g. feature calls and contracts, to shield programmers from low-level concurrency concepts such as threads, mutexes, and semaphores, letting them instead produce concurrent applications along the same lines as sequential ones.

The focus of my work is resolutely practical: I want to give programmers a simple but powerful tool for developing software. I base my work on Eiffel [92]; just like Eiffel is not only a language but a full methodology, I want SCOOP to be a methodology for building high-quality concurrent applications. Therefore, the "tool" becomes a complete *programming framework* consisting of several parts:

- A computational model
- An extension of Eiffel to support the model
- A compiler and supporting libraries
- Teaching material

The computational model should be based on object-oriented principles; it should integrate *seamlessly* all object-oriented mechanisms and take advantage of their power, rather than constraining them. Therefore, no typical "concurrency on top of O-O" or "O-O on top of concurrency" solution is acceptable where either O-O concepts or concurrency mechanisms are only partially supported and there exist a number of "special" rules for the concurrent case. One may expect that a full integration of O-O and concurrency will result in the absence of such rules; I take it to be the primary evaluation criterion for this part of my work. Another important criterion is the support for all O-O mechanisms: there should be no restrictions on the use of genericity, inheritance, polymorphism, dynamic binding, and agents. The formal model should also be a good platform for *reasoning* about software. Design by Contract enables modular reasoning; so should SCOOP.

The language extension should be minimal, both in terms of the annotation burden put on programmers, and the impact on the underlying sequential language. The extension should introduce no ambiguity or inconsistency; the number of additional keywords should be kept to a minimum. Syntactic constructs should correspond clearly to the concepts of the underlying computational model, so that programmers can easily express their designs. Conversely, the language should enforce clarity in the design. Since the language strives for abstraction and convenience, it must hide low-level details irrelevant to most programmers. Nevertheless, if the access to such details is necessary, it should be supported through libraries.

A language is only as good as its supporting tools. A compiler is necessary to translate SCOOP programs to executable code. This can be done either directly or indirectly, e.g. by translating SCOOP to pure Eiffel and then relying on an existing compiler. The main evaluation criterion for the tools is the extent to which SCOOP constructs are supported; compatibility with existing Eiffel tools and libraries is also essential.

The last component of the framework — teaching material — might come as a surprise but I believe that no model or methodology may ever be successfully used in practice, or claimed to be simple, if it is difficult to learn. Therefore, I have decided to teach SCOOP in a graduate course on concurrent programming at the ETH Zurich. Since the course participants have very different backgrounds in terms of industrial experience and previous use of other concurrency mechanisms, but none of them has used SCOOP before, the course is a good testbed for the framework. I take the results of the course and students' feedback to be important evaluation criteria for simplicity and usability of SCOOP and its tools.

1.3 Conventions

This dissertation describes my own work but — research being a collective activity — many of the presented ideas are a result of discussions and cooperation with other people. Discussion sections acknowledge the contributors and give references to existing work.

This document is intended to be self-contained. Nevertheless, a good command of object-oriented concepts and Design by Contract, as described in Bertrand Meyer's book "Object-Oriented Software Construction" (OOSC2) [94], is a prerequisite for understanding the dissertation.

The dissertation uses the nomenclature introduced in OOSC2 and the recent ECMA/ISO Eiffel standard [53, 68]. The naming conventions differ somewhat from those of Java, C++, and C♯. A glossary of basic terms is included in appendix B. The BON notation [145] is used in class diagrams. The notation for object diagrams is extended with SCOOP-specific elements. Diagrams include a key; in a series of related diagrams shown in the same chapter, only the first one has a key.

Writing "the original SCOOP model" and "the current SCOOP model" in technical discussion is confusing and burdensome. From now on, I will refer to the original SCOOP model as *SCOOP_97*, and to the current model developed in this dissertation simply as *SCOOP*. Chapter 4 gives a precise justification for this convention.

"O-O" is used as shorthand for "object-oriented". I also use "DbC" as shorthand for "Design by Contract", and "COOL" as shorthand for "concurrent object-oriented language". For brevity, I use words such as "he" and "his", in reference to unspecified persons, as shortcuts for "he or she" and "his or her", with no connotation of gender.

Although I have written the present chapter in the first person singular, I use the plural form "we" in the rest of the dissertation. This enables putting myself in the background and avoiding awkward changes of style between sections dealing with my own work and those describing joint work with other people.

2

Summary and main results

THIS chapter outlines the approach, defines the scope of the dissertation, and describes its main results and contributions.

2.1 Summary

This dissertation presents SCOOP: a practical framework for the development of high-quality concurrent software. The framework carries the advantages of object technology and Design by Contract to the concurrent context. We put an emphasis on the practical aspects of the methodology: it is simple, expressive, and well-supported by tools. It achieves simplicity by relying on the basic O-O concepts; its expressive and modelling power is due to the full support for advanced O-O mechanisms and DbC. Abstraction and encapsulation enable modular design and analysis of programs, which results in good scalability. Unlike most existing COOLs, SCOOP is a full-blown O-O language: it supports (multiple) inheritance, polymorphism, dynamic binding, genericity, and contracts. It comes with a compiler and a set of libraries. We also deliver teaching material in the form of lecture slides, exercises, and examples.

We consider the SCOOP_97 model introduced by Bertrand Meyer in [93, 94] to be a convenient basis for our study. It is a good candidate for modelling object-oriented concurrent applications because it relies on powerful object-oriented principles, and it tries to take advantage of Design by Contract. The framework presented in this dissertation should be seen as a further development of Meyer's model, although we take a different approach with respect to the relation between sequential and concurrent programming. The original model was an attempt at finding the smallest step from the sequential to the concurrent world. We agree with that approach as far as *reasoning* about concurrent software is concerned. A human brain is skilful at sequential reasoning about small portions of code but it cannot deal with a large number of complex parallel processes. Therefore, we try to make it possible to reason about concurrent programs in a sequential and modular way. For the *semantics*, we take the opposite view: systems are concurrent in general, and sequentiality is just a special case of concurrency. Following this argument, we look for a unified, generalised semantics of certain object-oriented mechanisms that is applicable in a concurrent context, and show that sequential programming relies on a specialised version of that semantics. In a way, we try to find the smallest step from the concurrent world to the sequential world.

We start with an in-depth analysis of SCOOP_97 to identify its inconsistencies and limitations. We propose adequate solutions to the encountered problems, extend the model, formalise it, and provide an implementation. A number of steps are followed:

- Analysing the relationship between object-orientation and concurrency, with a particular focus on the role of assertions — preconditions, postconditions, invariants, and loop assertions — in the concurrent context. We show that Design by Contract is beneficial for concurrent systems in that it supports specifying all the required conditions — including the appropriate synchronisation — for a correct interaction between clients and suppliers. We generalise the meaning of contracts so that assertions are evaluated asynchronously whenever possible, while keeping their contractual character. Condition synchronisation is based entirely on preconditions and postconditions. We demonstrate that the new semantics boils down to the standard sequential semantics when no concurrency is involved.

- Exploring the feasibility of modular reasoning about concurrent software. We propose a Hoare-style rule which unifies the treatment of synchronous and asynchronous feature calls, and enables modular and sequential-like reasoning about them.

- Refining the consistency rules and integrating them with Eiffel's type system. The enriched type system captures the concurrency-related properties of entities: their locality and detachability; carefully designed type rules eliminate potential atomicity violations without restricting the expressiveness of the model.

- Refining the semantics of feature application and argument passing mechanisms to support selective locking and lock passing which increase the expressiveness of the model and optimise the use of computational resources.

- Providing a full support for the advanced object-oriented mechanisms: genericity, polymorphism and dynamic binding, agents, and once features.

- Implementing the model and the supporting tools: the core (*SCOOPLI*) library and the (*scoop2scoopli*) compiler which type-checks SCOOP code and translates it into pure Eiffel code with embedded library calls to SCOOPLI. The supporting *CONCURRENCY* library provides advanced facilities: asynchronous agent calls, asynchronous handling of events, parallel waiting for multiple results, etc.

- Providing the teaching material for SCOOP: lecture slides, exercises, and examples. The material is tested in two iterations of a graduate course on concurrent programming.

The theoretical and practical results of our research help simplify the construction of concurrent systems by bringing the O-O programming method for such software to a higher level of abstraction and convenience, and making it easy to understand, learn, and apply.

2.2 Topics not covered in this dissertation

This dissertation focusses on the basic object-oriented concurrency mechanism of SCOOP, its type system, the support for advanced O-O techniques, and the implementation. A number of theoretical and practical topics fall beyond the scope of this work:

- *User-defined mapping of abstract concurrency resources to physical resources*
  SCOOP_97 proposes the CCF (Concurrency Control File [94]) as a means to decide what

physical resources (machines, CPUs, processes) each abstract thread of control (processor) should be mapped to. For the purpose of our study, we assume a single machine environment; our implementation maps each processor to a single thread (POSIX or .NET); the number of available threads is not bounded (see section 11.2).

- *Exception handling*
  Exception handling raises some particular problems in SCOOP due to the presence of asynchronous feature calls: an exception may be raised when a faulty client has already left the context where the corresponding call occurred, so that the exception cannot be propagated properly. We do not consider exceptions in our formal model. An extension of Eiffel's exception mechanism has been recently described by Arslan et al. [11]; the proposal tackles the issue of asynchrony. An earlier study [107] suggested a *wait on rescue* semantics to reduce asynchronous exceptions to synchronous ones.

- *Real-time programming*
  Real-time programming with SCOOP is a topic of another PhD project in our group [9]; that project also considers the duel mechanism proposed as part of SCOOP_97. Our recent article [10] describes the combination of SCOOP and event-driven programming for modelling real-time applications.

- *Distributed implementation*
  One of the goals of a general concurrency mechanism is to make the physical distribution of objects transparent to the programmer. In SCOOP, a number of mechanisms which facilitate distributed programming are provided, e.g. the creation of objects on a specified processor (see section 6.6) and the object import operations (see section 6.8). Nevertheless, our implementation only targets a single machine; a feasibility study of a distributed implementation is the topic of a separate project [121].

- *Proofs of deadlock-freedom*
  This dissertation develops the topic of modular proofs of safety properties; some liveness properties, such as loop termination, are also considered. We study the relation between deadlocks and contract violations (see chapter 8), and devise techniques that reduce the potential for deadlock (see chapter 7), but we do not provide any method for proving the absence of deadlock. We expect, however, that the model proposed here can be extended to cover deadlock prevention. A run-time mechanism for deadlock detection has been devised and implemented as an extension of the *SCOOPLI* library by Moser [100].

- *Operational semantics of SCOOP*
  We discuss properties of the type system proposed in this dissertation (see section 6.12) but do not provide a formal justification of its soundness. A formal study of SCOOP, including the development of a full operational semantics and a proof of type soundness, would constitute a rich PhD topic in itself. A variant of fair transition systems [88] could be used as basis for the operational semantics. Ostroff et al. [114] propose such a model for a subset of SCOOP_97, to support reasoning about program properties beyond contracts.

2.3 Organisation of the dissertation

This dissertation may be read sequentially from cover to cover, or selectively. Here is a brief description to facilitate the navigation:

- The rest of the current chapter presents the main contributions of this dissertation.
- Chapter 3 discusses related work on object-oriented concurrency models and languages, concurrency in Eiffel, and other work relevant to our research.
- Chapter 4 presents the original SCOOP_97 model proposed in [93, 94]. It includes a historical overview of its development and discusses the subsequent work on the model by other authors.
- Chapter 5 is a critique of the original model. It serves as a roadmap for the development of the current framework; all the identified issues are addressed in chapters 6 – 10.
- Chapter 6 discusses the computational model of SCOOP and introduces an enriched type system for safe concurrency.
- Chapter 7 presents a refined access control policy: relaxed locking rules and a lock-passing mechanism.
- Chapter 8 proposes a generalised semantics of contracts and a proof technique for asynchronous and synchronous feature calls.
- Chapter 9 discusses the support for advanced O-O mechanisms: multiple inheritance, polymorphism and dynamic binding, genericity, agents, and once features.
- Chapter 10 discusses the practical applications of SCOOP; several examples illustrate the discussion.
- Chapter 11 describes the implementation and the supporting tools; various implementation issues are discussed.
- Chapter 12 discusses the experience with a SCOOP-based graduate course on concurrency.
- Chapter 13 assesses the work presented in this dissertation, describes its limitations, and points out possible directions for future research and development.

2.4 Main results and contributions

SCOOP model and language

We start with a description and a detailed critique of the existing SCOOP_97 model, and continue with a development of current SCOOP. The computational model and the language extension described below are derived from SCOOP_97 but we refine them to eliminate the identified inconsistencies and limitations.

Computational model

Concurrency in SCOOP relies on the basic mechanism of object-oriented computation: the feature call. Each object is handled by a *processor* — a conceptual thread of control [1] — referred to as the object's *handler*. All features of a given object are executed by its handler, i.e. only one processor is allowed to access the object. Several objects may have the same handler; the mapping between an object and its handler does not change over time. If the client and supplier objects are handled by the same processor, a feature call is synchronous; if they have different handlers, the call becomes asynchronous, i.e. the computation on the client's handler can move ahead without waiting. Objects handled by different processors are called *separate*; objects handled by the same processor are *non-separate*. A processor, together with the object structure it handles, forms a sequential system. Therefore, every concurrent system may be seen as a collection of interacting sequential systems; conversely, a sequential system may be seen as a particular case of a concurrent system (with only one processor).

Since each object may be manipulated only by its handler, there is no object sharing between different threads of execution (no shared memory). Given the sequential nature of processors, this results in the absence of intra-object concurrency, i.e. there is never more than one action performed on a given object. Therefore, programs are data-race-free by construction. We use locking to eliminate *atomicity violations*, i.e. illegal interleaving of calls from different clients.

For a feature call to be valid, it must appear in a context where the client's processor holds a lock on the supplier's processor (i.e. the supplier is *controlled* by the client; see section 6.5). Locking is achieved through the refined mechanism of feature application: the processor executing a routine with attached formal arguments blocks until the processors handling these arguments have been locked (atomically) for its exclusive use; the routine serves as critical section. Since a processor may be locked and used by at most one other processor at a time, and all feature calls on a given supplier are executed in a FIFO order, no harmful interleaving occurs. Condition synchronisation relies on preconditions: a non-satisfied precondition causes waiting. Clients resynchronise with their suppliers if and when necessary, thanks to *wait by necessity* [38]: clients wait only on queries (function or attribute calls); commands (procedure calls) do not require any waiting because they do not yield results that clients would need to wait for.

Language extension

The language extension supporting the model is minimal; SCOOP only needs to enrich Eiffel with type annotations which express the relative locality of objects represented by entities and expressions. An entity may be declared as one of:

- x: X
  Objects attached to x are handled by the same processor as the current object. We say that x is *non-separate* with respect to **Current**.
- x: **separate** X
  Objects attached to x may be handled by any processor; this processor may (but does not

[1] See section 4.2.1 for a formal definition. A processor is an abstract concept: it does not have to be associated with a physical CPU; it may also be implemented by a process of the operating system, or a thread in a multithreading environment. Here, we assume the latter implementation.

have to) be different from the one handling the current object. We say that x is *separate* from **Current**.

- x: **separate** $<p>$ X
  Objects attached to x are handled by a processor known under the name 'p'. The processor tag 'p' may have an unqualified form and be explicitly declared as p: *PROCESSOR*, or have a qualified form derived from the name of another entity, e.g. 'y.**handler**' (in which case y must be an attached read-only entity, e.g. a formal argument). We say that x is *separate* from **Current**, and handled by 'p'.

The first two options were already present in SCOOP_97; the third one is new. Additionally, a type annotation may include the '?' sign, e.g. x: ?**separate** X; this marks a detachable type [96], i.e. the decorated entity may be void (not attached to any object) at run time. Entities not decorated with '?' are attached, i.e. not void.

Improvements on the previous model

Our model improves on SCOOP_97 in a number of ways. The main contributions are:

- *Precise semantics of the feature call mechanism*
  We introduce a distinction between the *feature call* and the *feature application* mechanisms, to define clearly the duties of clients and suppliers in the object-oriented computation (see section 6.1). These mechanisms were incorrectly amalgamated in the earlier model; this complicated the validity and consistency rules, and hindered the understanding of other essential mechanisms, e.g. argument passing and contracts. The new semantics of feature calls simplifies the validity rules and the synchronisation mechanism of SCOOP; the generalised semantics of contracts (see section 2.4) relies on the introduced distinction.

- *Clarification of the separate semantics*
  SCOOP_97 uses two different semantics for the **separate** keyword. Some rules follow the *strict* semantics whereby separate entities denote objects handled by a processor that is *necessarily* different from the one handling **Current**; other rules allow a *non-strict* interpretation whereby separate entities denote *potentially* separate objects. We apply the non-strict semantics uniformly across all the rules and mechanisms of the language. Consequently, the declaration of separate classes, i.e. classes whose instances are separate with respect to to all other objects, is prohibited; this eliminates a source of inconsistency (see the *Separate Current paradox* in section 5.1) and greatly simplifies the type system.

- *Integration of validity rules with the type system*
  We enrich Eiffel's type system with *processor tags* which precisely capture the relative separateness of objects (see section 6.2). The **separate** keyword loses its special status; it becomes a mere type annotation. SCOOP_97's consistency rules turn out to be superfluous: after the necessary refinement — to make them sound yet less prohibitive — they are subsumed by our type rules. The integration of informal rules into the formal type system largely simplifies the model. SCOOP unifies the treatment of separate and non-separate calls: all the rules of the model apply to both kinds of calls.

- *Precise reasoning about object locality*
  While SCOOP_97 only supports specifying the separateness of an object with respect to **Current**, SCOOP's enriched types permit a precise reasoning about object locality. Processor tags are used to assert that several objects are non-separate with respect to each other (even if they are separate from **Current**); this allows safe attachments and feature calls between these objects without the need for additional locking. It is also possible to create objects on a chosen processor.

- *Increased expressiveness of the model*
  SCOOP offers more flexibility in the use of separate calls (see section 4.2.4). Syntactic restrictions on the targets of separate calls disappear; a new call validity rule enables calls on separate targets other than formal arguments. As a result, it is now possible to build multi-dot expressions involving separate entities, and use attributes and local variables as targets. Additionally, all restrictions on the use of expanded types go away. The increased expressiveness is due to the enriched type system and the new semantics of contracts (see section 2.4).

- *Flexible locking*
  We optimise the locking policy so that only the necessary locks are acquired. Type-based locking rules using the new semantics of *attached types* make it possible to decide which formal arguments of a routine should be locked. (SCOOP_97 requires all arguments to be locked, whether it is necessary or not.) A lock passing mechanism permits clients to pass their locks to a supplier for the duration of a single call. The proposed mechanism makes concurrent programs less deadlock-prone and allows the implementation of interesting synchronisation scenarios, e.g. separate callbacks and cross-client locking, without violating the atomicity guarantees (see chapter 7 for details).

- *Support for full asynchrony*
  SCOOP_97 supports asynchronous calls but any sequence of such calls appearing in a routine body has to be preceded by a synchronisation event caused by the call to that routine and the resulting locking. Our framework enables fully asynchronous calls (see section 9.3.4). A client issuing such a call immediately continues its activity; the call will be executed on the client's behalf at some point in the future. There is no assumption on the execution delays, but consecutive calls on the same target are guaranteed to be applied in the FIFO order. This mechanism solves, among others, the problem of parallel wait (referred to as "waiting faster" by Tony Hoare [66]); a client is now able to spawn several computations in parallel and wait for the first result (see section 10.2.2).

New semantics of contracts

A major contribution of this dissertation is the generalisation of Design by Contract to concurrency, to take advantage of the modelling power of DbC and to make a full use of assertions in the proofs of asynchronous calls (see chapter 8). We analyse the impact of concurrency on each type of assertion and propose a generalised semantics that applies to both the concurrent and the sequential context. We demonstrate the interplay of the new semantics with other O-O techniques and mechanisms; we also show that the standard correctness semantics used in the sequential Eiffel is a refinement of the new one.

The original SCOOP_97 model uses two different semantics for preconditions, depending on whether they involve separate calls: separate preconditions have wait semantics, i.e. a violated precondition causes the client to wait, whereas non-separate ones are correctness conditions, i.e. a non-satisfied precondition is a contract violation and results in an exception. This is confusing because the same syntactic construct — the **require** clause — is used for two different purposes. No attempt has been made at exploring the potential of other assertions; they simply keep their sequential semantics. Separate postconditions are particularly problematic: they cause waiting, thus minimising the potential for parallelism and increasing the danger of deadlock. Additionally, separate assertions are completely excluded from proof rules (see section 5.7), which complicates the formal reasoning about software.

The new semantics of assertions proposed in SCOOP solves these problems. No distinction is made between separate and non-separate assertions; all of them preserve their contractual character and they may be used for reasoning about concurrent programs, provided that they are *controlled*, i.e. only involve calls on targets which are already locked by the client in the context of the call. The unified semantics provides a sound support for polymorphism, feature redefinition, and dynamic binding.

- All preconditions now have the *wait semantics*: before executing a routine, the executing processor waits until the precondition is satisfied. In the case of a violated controlled precondition, however, waiting is only conceptual: since the client controls all the involved objects, it can be immediately blamed for not establishing the precondition.

- Postconditions keep their usual meaning — they describe the result of a feature application — but each postcondition clause is evaluated individually and asynchronously; a client does not wait unless its handler is involved in a given clause. This increases the amount of concurrency without compromising the guarantees given to the client.

- Loop assertions and check instructions follow a similar pattern as postconditions: they are evaluated without forcing the client to wait, while still delivering the required guarantees. On the other hand, the individual evaluation of subclauses does not apply; the whole assertion has to hold at the same time even if it involves multiple handlers.

- Class invariants keep their usual semantics because asynchronous calls are prohibited in invariants. This is not imposed by any explicit rule for separate assertions but follows from the refined call validity rule.

We demonstrate that the new semantics is indeed a generalisation: in the absence of separate calls it simply reduces to the usual Eiffel semantics. The proposed extension of DbC clarifies the model and eliminates the overloading of several concepts and language constructs. It contributes largely to the better understanding and support of other O-O mechanisms, e.g. feature call, polymorphism, and dynamic binding. It also facilitates the sequential-to-concurrent and concurrent-to-sequential code reuse because contracts have the same unambiguous meaning in both contexts (see chapter 8). Most importantly, all assertions can now be used for modular reasoning about programs; we remove the syntactic restrictions from the proof rule for feature calls (see section 2.4).

Type system

SCOOP_97 has a number of consistency rules to prevent the occurrence of *traitors*, i.e. non-separate entities that represent separate objects. A traitor may cause atomicity violations because its clients are able to perform separate calls without respecting the mutual exclusion policy. The existing rules are too weak to eliminate all potential traitors; at the same time, they are too prohibitive: many useful (and safe) programs are rejected (see section 5.4). In particular, it is impossible to perform calls on multi-dot expressions that involve separate entities. Furthermore, the use of expanded types is restricted: non-fully-expanded objects cannot be passed as actual arguments or results of separate calls.

The analysis of the informal rules reveals that they simply try to capture a *conformance relation* between separate and non-separate entities, and to prohibit operations that do not respect the conformance relation, e.g. assignments from a separate to a non-separate entity should be prohibited, whereas assignments in the opposite direction are permitted. In SCOOP, we take this effort one step further by refining the intended conformance rules to ensure their soundness, relaxing the unnecessary restrictions, and integrating the rules with an enriched type system.

Enriched types

We extend Eiffel's type system with *processor tags* which specify the relative locality of objects. A SCOOP type *T* is represented as a triple (γ, α, C) with the following components:

- *Detachable tag* $\gamma \in \{!, ?\}$
  A type is either attached ($\gamma = !$) or detachable ($\gamma = ?$), in the standard Eiffel sense: entities of an attached type are guaranteed to be non-void at run time; detachable entities may be void [96, 53].
- *Processor tag* $\alpha \in \{\bullet, \top, p, \bot\}$
  The processor tag captures the locality of objects represented by an entity of type T, i.e. their separateness or non-separateness with respect to other objects. '$\bot$' denotes *no processor*; it is used to type **Void**. If there is an object, it is one of: non-separate, i.e. handled by *current processor* ('$\bullet$'); potentially separate, i.e. handled by *some processor* ('$\top$'); handled by the processor *p* ('p').
- *Class type* C
  This is the traditional Eiffel type, based on a simple class, e.g. *X*, or its generic derivation, e.g. *LIST* [*X*].

In the program text, types are specified using the extended syntax outlined in section 2.4. For example, an entity declared as

x: X

has the type $(!, \bullet, X)$; a declaration

x: **separate** X

yields the type $(!, \top, X)$, and a declaration

x: ?**separate** <*py*> X

yields the type $(?, py, X)$.

Subtyping

The subtyping relation is based on the conformance relations of the three type components. The conformance of class types is based on inheritance (with additional rules for expanded types and generic classes) just like in sequential Eiffel. Attached ('!') conforms to detachable ('?'). Processor tags are ordered in a lattice, with the top element '$\top$' and the bottom element '$\bot$'; other tags conform to '$\top$' but not to each other. For example, a type $T_2 = (!, \bullet, Y)$ is a subtype of $T_1 = (?, \top, X)$, provided that Y inherits from X; similarly, $T_3 = (!, \top, X)$ is a subtype of T_1. The validity rules for assignment, feature call, object creation, etc. follow this subtyping relation. For example, the assignments from an entity of type T_2 or T_3 to an entity of type T_1 are valid, whereas the assignments in the opposite direction are invalid because the types do not conform.

Thanks to the enriched type system, all potential atomicity violations are detected and eliminated at compile time. The type system is sound and much more flexible than SCOOP_97 rules; it gives programmers more freedom in expressing interesting synchronisation scenarios. Since static types are always an approximation of run-time types, we also extend the *object test* mechanism to support safe downcasts between types with different processor tags (see section 6.7).

Attached types

Attached types — proposed in [96] and later adopted in the Eiffel standard [53] — are an essential tool for eliminating calls on void targets. Our type system takes the detachability of entities into account but does not eradicate such calls. The formal rules introduced in section 6.11 ensure that a correctly initialised attached entity remains attached and never gives raise to a void target call; the initialisation rules, however, are not formalised. We use attached types to refine the semantics of argument passing and locking so that routines only lock their attached formal arguments. A new call validity rule relies on this refinement; it eliminates calls on unlocked targets. In figure 2.1, the attached type of the formal argument x indicates that the routine acquires x's handler before executing the body; the handlers of y and z are not locked because both entities are declared as detachable. Calls on x are permitted in the body of r, whereas calls on y and z are not. (But nothing prevents the use of y or z as sources of attachments: argument passing or assignment.)

```
r (x: separate X; y, z: ?separate Y)
    do
        x.f
        my_y := y
        x.g (z)
    end
```

Figure 2.1: Selective locking based on attached types

Programmers can use attached types to indicate which formal arguments should be locked; this results in an optimal control of resources and minimises the danger of deadlock. The refined semantics of argument passing enables passing the locks from clients to their suppliers

for the duration of a single call (see section 7.2). This mechanism is necessary to enable several useful synchronisation scenarios, e.g. cross-client locking and separate callbacks, that could not be implemented in SCOOP_97. Not only does the lock passing mechanism increase the expressiveness of SCOOP while preserving all its safety guarantees, it is actually necessary for sound reasoning about asynchronous calls (see section 8.2).

Expanded types

Expanded types are used to express an ownership relation between objects (e.g. a car engine *belongs* to a given car), and to emulate unique references. An expanded entity, that is an entity whose type is based on an expanded class, represents an object rather than a reference to an object; expanded objects are always passed by copy. SCOOP_97 only allows *fully expanded* objects as arguments of separate calls. A fully expanded object must not carry any non-separate references; in practice, this restricts the choice of expanded classes to *BOOLEAN*, *INTEGER*, *REAL*, *DOUBLE*, and *CHARACTER*. We refine the type system of SCOOP to accommodate arbitrary expanded types — in particular user-defined ones — without breaking the safety guarantees (see section 6.10).

Support for advanced object-oriented mechanisms

Important O-O concepts, e.g. genericity, polymorphism, and agents, have not been studied in SCOOP_97; other mechanisms, e.g. once functions, are only partially supported there. The development of our framework has been driven by the need to provide a full support for the O-O methodology, including its advanced concepts. Multiple inheritance, genericity, polymorphism, feature redefinition, precursor calls, once features, and agents are seamlessly integrated into SCOOP (see chapter 9).

- *Genericity*
  Enriched type annotations may appear in actual and formal generic parameters; their use is only limited by the type conformance rules. For example, it is now possible to declare a list of potentially separate objects

  l : *LINKED_LIST* [**separate** *BOOK*]

  There is no limit to the nesting of separate generic parameters; enriched type annotations may appear in arbitrarily nested generic types, e.g.

  nested_gen : *A* [**separate** <*px*> *B* [*C*, **separate** *D* [**separate** *E* [...]]]]

  Constrained genericity also makes use of the enriched type system; new type annotations may appear in constraints, e.g.

  class *MY_CONTAINER* [*G* −> **separate** *ELEMENT*]

  The strengthened conformance rule 9.2.3 for generically derived class types, which allows for limited covariant subtying — *B* [*U*] is a subtype of *A* [*T*] if and only if *B* conforms to *A*, *U* conforms to *T*, and *U* is either detachable or identical with *T* — closes a loophole in Eiffel's type system.

- *Polymorphism, dynamic binding, feature redefinition*
  A client must not be cheated upon in the presence of polymorphism and dynamic binding, i.e. the actual version of a feature chosen at run time must abide by the original contract known to the client at compile time. The usual DbC rules for assertion redefinition, which enable precondition weakening as well as postcondition and invariant strengthening, satisfy this requirement in the concurrent context. A weaker precondition results in (potentially) less waiting; a client never waits longer than with the original contract. A strengthened postcondition gives stronger guarantees but does not imply any additional waiting on the client's side. Invariants cause no waiting so their strengthening does not influence it.

  The rules for argument and result types of features need to take into account the enriched notion of type, while remaining compatible with the standard Eiffel rules. Detachable tags may be redefined from '?' to '!' in result types, and from '!' to '?' in argument types. Given the locking semantics of attached types, a redefined version of a feature may lock at most as many arguments as the original one. In other words, a client may expect at most as much locking as specified by the original signature. Similarly, processor tags may be redefined from the most general '$\top$' to something more specific in result types, and in the opposite direction (from more specific to '$\top$') in argument types. The combined rules for processor tags and detachable tags satisfy Liskov's substitution principle [84]. Nevertheless, the argument redefinition rules raise one issue: the inherited precondition and postcondition clauses that involve calls on the redefined formal arguments may become invalid. Therefore, we disallow the redefinition of a formal argument from attached to detachable if the inherited postcondition involves calls on that formal argument. (This effectively eliminates the problem of potential postcondition weakening, not considered in the Eiffel standard [53].) No such restrictions are put on arguments involved in preconditions: if an inherited precondition clause involves a call on a detachable argument, it is considered to hold vacuously; this weakens the precondition, which is compatible with the rules of DbC.

- *Agents*
  Agents are integrated into our model and type system in a straightforward manner. We place an agent on the same processor as its target; consequently, the processor tags of the agent and its target are identical. This ensures the safe use of agents without any special rules; agents are treated just like any other object. A call on an agent — which is effectively a call on its target — is only allowed in a context where the agent's handler is locked; since it is also the target's handler, no atomicity violation occurs.

  The refined agent mechanism provides a convenient way to represent partially or completely specified asynchronous computations as first-class citizens of the object-oriented world. A number of advanced facilities rely on agents:

  – Fully asynchronous calls.
  – Parallel waiting ("waiting faster") on several activities.
  – Elimination of burdensome dummy routines used for wrapping single separate calls. A generic enclosing routine may be used instead.

  All these mechanisms are implemented as part of the CONCURRENCY library delivered with SCOOP.

- *Once functions*
  We clarify the semantics of once functions to allow their safe use in a concurrent context. A once function of a separate type has *once per system* semantics, i.e. its result is shared by all the instances of the declaring class, no matter what processors they are handled by. A once function of a non-separate type has *once per processor* semantics, i.e. its result is shared by all instances of the class handled by the same processor.

Reasoning about concurrent programs

We propose the Hoare-style rule 2.4.1 for reasoning about synchronous and asynchronous feature calls; it can be used for proving safety properties of programs but it is also strong enough to prove certain liveness properties, e.g. loop termination. We do not attempt to develop a full proof system for SCOOP; nevertheless, our effort may be seen as a first step in that direction.

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{Pre_r^{ctr}[\overline{a}/\overline{f}]\}\ \ x.r(\overline{a})\ \ \{Post_r^{ctr}[\overline{a}/\overline{f}]\}} \tag{2.4.1}$$

The proof technique, described in section 8.2, is derived from the sequential technique but based on the new contract semantics. It unifies the treatment of synchronous and asynchronous calls. Preconditions and postconditions involving separate calls are not discarded (as in SCOOP_97), provided that the involved entities are *controlled*, i.e. attached and locked in the context of the call under scrutiny. Such assertions are referred to as *controlled clauses* (see definition 8.1.2); hence the superscript ctr decorating them in the conclusion of the proof rule.

Our approach is novel in that it eliminates the need for a special treatment of asynchrony. Sequences of asynchronous calls — or interleaved synchronous and asynchronous calls — may be reasoned about using only the preconditions and the postconditions, without the need for temporal operators. (Our earlier approach [113] relied on temporal logic, which resulted in more complex rules.) Thanks to their asynchronous semantics, postconditions may be projected in the future, i.e. assumed *immediately* after the call even though they will only be established *eventually*. Other assertions — class invariants, checks, loop variants and invariants — are used in the same way as in a sequential context; they are assumed to hold immediately even if their evaluation may be delayed through the involved asynchronous calls. Combined with the new asynchronous semantics, it opens new opportunities for exploring the potential parallelism without sacrificing the safety guarantees. For example, loops involving asynchronous calls can be proved correct: loop assertions are simply registered to be evaluated asynchronously in a correct order; a client does not need to wait but it gets the safety guarantees stated in the loop invariant and the termination guarantee ensured by the variant (see section 8.1.4).

Our technique is limited to reasoning about the properties of controlled entities. The absence of deadlocks caused by indefinite waiting on uncontrolled preconditions or a non-available actual argument cannot be proved; non-modular reasoning using temporal logic is necessary in such situations, to take into account the interference of other clients (see section 8.2).

Code reuse

Popular languages which support multithreading, e.g. Java and C♯, allow a limited reuse of sequential code in concurrent programs. A naive reuse of library classes that have not been

designed for concurrency often leads to data races and atomicity violations. The supplier-side (server-side) synchronisation policy applied in these languages requires the classes used in a concurrent application to be written "with concurrency in mind", i.e. any feature that may potentially be accessed simultaneously by several threads needs to implement an appropriate synchronisation mechanism, usually in the form of a

while (*someCondition*) *wait*();

loop. It is difficult to guess all the future contexts in which a class may be used; therefore, programmers often implement libraries in a defensive style, providing additional safety mechanisms just in case. This results in heavy, entangled code which is difficult to extend and reuse.

SCOOP eases the reuse of sequential libraries. The mutual exclusion guarantees offered by the model make it possible to assume a correct synchronisation of clients' calls and focus on solving the problem without bothering about the exact context in which a class will be used. This leads to a clearer code which can be reused and extended through inheritance. A sequential class may be taken and used in a concurrent application with no need for modifications. For example, the class *QUEUE* [*G*] may represent a shared buffer; it suffices to declare an appropriate entity, e.g.

buffer : **separate** *QUEUE* [*INTEGER*]

There is no need to provide a specialised version of the class equipped with additional synchronisation code.

Unlike many other approaches, including the multithreaded models of Java and C♯, SCOOP supports a concurrent-to-sequential reuse, i.e. the code written for a concurrent application can be safely used in a sequential context. This may seem obvious at first — according to the thesis established in this dissertation, sequentiality is just a special case of concurrency — but it is not a trivial problem, in particular in the presence of condition synchronisation. For example, a Java class implementing a shared buffer may not work properly in a sequential context because a thread performing the operation *put* may be blocked if the buffer is full, and wait for other threads to wake it up; it will deadlock if no other thread ever accesses the buffer. In SCOOP, the same class *BUFFER* may be used in both context, although it has been written primarily for a concurrent application. If the buffer is only used by one (non-separate) client — just like in the Java example with a single thread — the precondition semantics nicely reduces to the correctness semantics. As a result, an attempt at storing an element in the full buffer violates the contract; the client is given the opportunity to handle the erroneous situation rather than deadlocking. Concurrent code, when used in a sequential context, behaves just like sequential code; this is the essence of concurrent-to-sequential reuse achieved in SCOOP.

Implementation

The implementation of SCOOP is a major contribution of this work. It is essential for validating the model and demonstrating its practicality. In fact, our work began with an attempt at implementing SCOOP_97. This effort has driven the research and revealed several limitations and inconsistencies within SCOOP_97, prompting us to redesign the model and finally leading to the development of the current SCOOP framework. Most theoretical and practical issues discussed in this dissertation have been uncovered during the implementation work.

We provide three tools:

- $SCOOPLI$
  A library which implements the basic model: processors, separate calls, new semantics of assertions, wait by necessity, atomic locking, and scheduling.

- $scoop2scoopli$
  A compiler which type-checks SCOOP code and translates it into pure Eiffel with embedded calls to SCOOPLI.

- $CONCURRENCY$
  A library which implements advanced concurrency features: generic enclosing routines, fully asynchronous calls, parallel wait for several concurrent activities, asynchronous event handling.

SCOOP has been implemented as an Eiffel library rather than a compiler extension. We focussed on concurrency issues right from the beginning, without getting bogged down in the intricacies of existing compilers. Also, at the outset of this study, no satisfactory open-source Eiffel compiler was available [2]. SCOOPLI targets multi-threaded platforms: POSIX and Microsoft .NET; it is compatible with all Eiffel compilers that support the EiffelThread library. The scheduling policy ensures strong fairness guarantees: the FIFO ordering of calls on the same target, and the absence of starvation. SCOOPLI relies heavily on the agent mechanism: each separate call is wrapped in an agent and passed to the supplier's handler; assertions are also represented as agents.

SCOOPLI provides all the basic concurrency mechanisms but its manual use would be burdensome because the necessary agent wrapping and explicit calls to the scheduler obscure the syntax. Therefore, the scoop2scoopli tool translates SCOOP programs into pure Eiffel with embedded calls to SCOOPLI features, so that programmers do not need to deal directly with the library. The tool was first conceived as a pre-processor but later a full type-checker was added; therefore, we view it as a full SCOOP-to-Eiffel compiler. It may be used as a command-line tool or be integrated with existing IDEs (so far, it has been integrated with EiffelStudio [105]).

The CONCURRENCY library provides a set of utility classes supporting advanced concurrency features; programmers may use these classes directly in their code, e.g. through inheritance. An agent-based mechanism for fully asynchronous calls is implemented. A parallel wait facility lets clients wait for one out of several computations spawned in parallel; a similar mechanism permits resource pooling, i.e. using one of several available resources. Furthermore, a generic enclosing routine is provided to eliminate the need for wrapping single separate calls in dedicated routines. Finally, the library extends the $EVENT\_TYPE$ class for event-driven programming [12] with concurrency facilities, supporting an asynchronous publication of events and an independent notification of multiple subscribed objects.

Several concurrent applications — ranging from the basic scenarios described in the OOSC2 book to GUI applications to controllers for a physical model of a double-shaft lift and a Lego MINDSTORMS™robot that sorts production items according to their colour — have been developed using our tools. See chapter 10 for details.

[2]The ISE Eiffel compiler became open-source in April 2006; see *http://eiffelsoftware.origo.ethz.ch/*.

Teaching

SCOOP comes with an extensive teaching material in the form of lecture slides, exercise sheets, and a rich library of examples. To test the practicality of our framework, we taught SCOOP in two iterations of a graduate course at ETH Zurich (*Concurrent Object-Oriented Programming*, a.k.a. *Concurrent Programming II*, course number 251-0268-00, summer semester 2005 and 2006). Since the course participants had different backgrounds in terms of industrial experience and previous use of other concurrency mechanisms, but none of them had used SCOOP before, the course was a good testbed for the framework. The course encompassed a historical overview of O-O concurrency, the basics of SCOOP, the use of O-O techniques and DbC in a concurrent context, advanced mechanisms (agents, real-time facilities), and a comparison of SCOOP with other models (multithreading, Ada tasking, active objects). The exercise sessions and the final project let students get acquainted with the practice of concurrent programming and use our framework to solve challenging concurrency problems.

See chapter 12 for a discussion of the course and the students' feedback. All lecture slides and exercise sheets are available on the course page

`http://se.ethz.ch/teaching/ss2006/0268`

Programming examples described in chapter 10 and many more can be found on the SCOOP project page

`http://se.ethz.ch/research/scoop`

3

Previous work

EXISTING object-oriented concurrency models and languages constitute the rich soil on which SCOOP has grown and matured. This chapter reviews previous work; several models are discussed and compared with SCOOP. We take a closer look at different techniques for the elimination of synchronisation defects, and discuss other important developments that have influenced our research. Finally, existing concurrent extensions of Eiffel are presented and compared with our model.

The discussion is limited to object-oriented concurrency; traditional approaches to concurrency have been described in many other articles and books. Basic concepts of concurrent and distributed programming are presented in an excellent book by Andrews [8]. Ben-Ari [22] gives a concise introduction to concurrency problems, enhanced with an overview of concurrent features in several programming languages. A general discussion of object technology falls outside the scope of this chapter. An in-depth study of O-O principles and Design by Contract can be found in the OOSC2 book [94]. The book contains numerous references to previous work; you may also browse the proceedings of related conferences, e.g. OOPSLA [1] and ECOOP [2].

3.1 Object-oriented concurrency models

Object-oriented approaches model things that exist concurrently in the real world; they should help programmers in thinking about programs which involve concurrent activities. The central problem here is to retain the advantages of conventional object-oriented programming while providing an efficient and flexible control over parallelism [131]. Object-oriented concurrency has been a dynamic field of research for the past two decades. This has resulted in a proliferation of different models and languages. Several surveys of concurrent object-oriented languages (COOLs) are available [118, 146, 122, 131, 123]. The thesis of Papathomas [118] gives a first precise classification of COOLs, focussing on the methods for combining objects with concurrency. Philippsen [123] provides an extensive survey of COOLs, covering more than 100 languages. He also considers low-level details — mapping of concurrency abstractions to the underlying hardware, object migration, scheduling of activities — which were omitted in Papathomas's work. Agha et al. [4] is a collection of papers describing main research directions in object-oriented concurrency.

[1] ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (http://www.acm.org/oopsla)

[2] European Conference on Object-Oriented Programming (http://www.ecoop.org)

Object paradigm

Briot et al. [30] elaborate on different ways in which the object paradigm is used in concurrent and distributed contexts. Three basic approaches are presented: *integrative*, *library-based*, and *reflective*. The integrative approach aims at integrating concurrency concepts into the object paradigm and offering the programmer a unified model. This is in contrast to the library approach which treats the object-orientation and the concurrency aspects as orthogonal. An example of library-based approach is the Concurrency Class for Eiffel [76] where messages are manipulated explicitly, using a set of mechanisms for asynchronous and remote communication. A more integrative approach, such as Eiffel// [37] blends message passing into a sequential language and hides the low-level details behind better abstractions. Reflective approaches rely on powerful reflection mechanisms to access and modify the meta-model of a concurrent system, e.g. the treatment of messages, scheduling, preemption, etc. SCOOP clearly follows the integrative approach: concurrency and object-orientation are fully blended. However, reflective mechanisms for various extensions, e.g. optimised scheduling, may be provided as part of the *CONCURRENCY* library described in section 11.4.

Briot et al. also define four object concurrency levels:

1. *Serial* (atomic): at most one feature is executed on a given object at a time. POOL-I, POOL-T [5], and Eiffel// [38] are among the languages implementing this strategy.

2. *Quasi-concurrent*: several feature activations may coexist but at most one is not suspended at any given time. This corresponds to a monitor-style synchronisation with condition variables [65]. Languages that implement this strategy include ABCL/1 [148], Concurrent Smalltalk [147], Java (with monitors and synchronised methods) [79], and C♯ [54, 46].

3. *Concurrent*: true intra-object concurrency is supported but some restrictions apply, e.g. the number of features executed in parallel is limited, only non-conflicting features are executed in parallel, etc. Most Actor-based languages [4], e.g. ACT++ [75], implement this strategy. CEiffel [86] also belongs to this category (see section 3.3).

4. *Fully concurrent*: concurrency within an object is not restricted. Usually, such objects are functional, i.e. they do not have volatile state. Several Actor languages support such concurrency through the use of so-called *unserialised objects*.

Our model belongs to the *serial* category: since each object may only be manipulated by its handling processor, and processors are sequential, concurrency is limited to at most one feature per object. (A refinement of the access control policy proposed in [110] supports safe interleaving of several features at the object structure level; this would place SCOOP in the *quasi-concurrent* category. However, the proposed extension has not been retained here due to its insufficient support for polymorphism.)

Classification of COOLs

Philippsen [123] provides the most complete COOL survey up to date (more than 100 languages). The paper discusses proposed designs of COOLs and classifies them according to several criteria:

- *Initiating concurrency*, i.e. how concurrent activities are created. There are five categories here: *automatic parallelisation*, *fork–join*, *cobegin*, *forall and aggregates*, *autonomous code*.
  Fork–join, cobegin, forall, and aggregates initialise parallel activities at arbitrary points of an otherwise sequential program. These mechanisms are targeted towards fine-grained parallelism. Autonomous-style initialisation, such as process declarations, active objects' bodies, and autonomous routines, make parallelism explicit. They are targeted towards coarse-grained parallelism with a few explicit activities. Our model belongs to the latter category, even though it does not support explicitly the notion of a parallel activity.

- *Coordinating concurrency*, i.e. the way in which parallel activities synchronise the access to shared resources. Existing languages may be split into two categories:

  – *Activity-centred coordination*, also known as client-side synchronisation.
    The burden of properly synchronising the access to a supplier object is put on the client. Various synchronisation mechanisms may be applied here, e.g. Conditional Critical Regions (CCRs), locks, semaphores, etc. The advantage of activity-centred coordination is its immunity to inheritance anomalies [90]. The obvious disadvantage is that the synchronisation code for the same supplier is spread over several places in the program. It is therefore difficult to enforce a proper synchronisation policy. If some client does not synchronise correctly, it may break the consistency of the supplier and invalidate other clients' actions.

  – *Boundary coordination*, also known as supplier-side (or server-side) coordination.
    The supplier object is responsible for ensuring the proper synchronisation of clients' accesses. Typical techniques used here include monitors, delay queues, method guards, enable sets, and **accept** statements in active objects' bodies. The Actor model [3] also falls into this category.

  Philippsen puts SCOOP_97 (referred to as "Meyer's proposal") in the boundary coordination category because he understands wait-conditions as method guards (which they are not: wait-conditions are specified on the client side and suppliers cannot enforce any additional waiting). SCOOP could be considered as an activity-centred coordination model because locking requirements and wait-conditions are specified in clients' code. Nevertheless, the mutual exclusion is enforced by the processor that handles the supplier, rather than by the clients themselves. Therefore, the boundary coordination style is also used. This subtle point has often been brought up in discussions with the multithreading community: one cannot simply say that SCOOP follows either the client-side or the server-side synchronisation strategy. We regard SCOOP_97 and SCOOP as hybrid approaches, with the mutual exclusion being enforced on the supplier's side, and atomicity and condition synchronisation specified on the client's side.

- *Locality*, i.e. mapping of objects and activities to computational resources (memory, CPUs). This aspect is particularly important in languages that target distributed execution platforms. Several approaches exist:

  – *Meta-level locality*
    Distribution is completely transparent. i.e. it is not possible to decide statically whether an object should be stored locally or on a remote processor. When a new

object is created, a meta-object assigned to the object's base class is consulted first. The meta-object processes all feature calls before passing them on to the actual supplier object. Therefore, it is possible to delegate all locality-relevant operations to the meta-object. The programmer is in charge of implementing an appropriate meta-class. Concurrency Class for Eiffel [76] is a good example (see section 3.3) .

- *External locality*
  Distribution of objects is beyond the scope of the language. The programmer is in charge of placing objects on different computing nodes and registering them with a name server, as well as retrieving references to these objects via the name server. Distribution is transparent but object migration is not supported. Java RMI [134] follows this approach.
- *Internal locality*
  Programmer can specify a node that should be used for the creation of a given object. If there is no explicit specification of the node, a standard distribution policy will apply. Distribution is transparent, except for a special syntax for object creation. Object migration may be supported with additional syntactic constructs. A default placement policy may be applied to the whole application except for parts of code that have to be tuned manually for a particular reason (efficiency, load-balancing, etc.). Several COOLs follow this approach: POOL [7], ABCL/1 [148], Guide [17].
- *Virtual topology/scope locality*
  Programmers use an abstract model of the computing system. Objects and activities are mapped to this model first; the abstract model is then mapped onto the physical topology. Three common techniques are used here: *abstract processor numbers*, *computational grid*, and *spaces*. Several languages follow this approach, e.g. Parallel C++ [26] and Distributed Eiffel [62]. Most languages in this category do not provide object migration facilities. Also, dynamic virtual topologies are rarely supported.
- *Group locality*
  The programmer explicitly forms clusters of objects that belong together. The runtime maps these clusters to the available physical resources. Object migration is transparent and can be used by the runtime to enhance performance, e.g. through load balancing. Distribution is fully transparent. An example of language with group locality is Emerald [73].

SCOOP processors may be viewed as a variant of scopes, where objects belonging to the same scope may call each other using synchronous calls, whereas objects in different scopes need asynchronous calls for communication. The proposal of a distributed SCOOP implementation [121] enables a direct specification of object placement through an appropriate library call. Such specifications work at two levels: programmers can specify a target processor for an object creation, and the physical location (machine) for the processor; there is no possibility to bind directly objects to machines. The implementation described in chapter 11 of this dissertation assumes a single machine with an unbounded number of threads; each processor is mapped to an individual thread.

Philippsen discusses other pertinent issues: parallel performance, broken encapsulation, inheritance anomalies, and expressivity of coordination constraints. He also summarises the pros and

cons of a full integration of concurrency with an object-oriented language vs. a library-based extension of an existing language. The conclusion of his survey is that key characteristics of object-oriented methodology or key performance factors of concurrency have to be sacrificed due to the difficulty of merging both paradigms. We do not agree with Philippsen; it is the lack of proper abstractions that causes the trouble.

Actors, active objects, inheritance anomalies

The *actor model* was introduced by Hewitt [64]. Initially, an actor was defined as an autonomous agent with intentions, resources, message monitors, and a scheduler. Later on, Hewitt proposed a more abstract model of concurrency based on causal relations between asynchronous events — sending or receiving a message — in different actors. The actor model was further developed by Agha [4]. An actor system consists of a set of concurrently executing actors and a set of messages in transit. Each actor has a unique name, which may be seen as an e-mail address, and an associated behaviour. Actors communicate using asynchronous messages. Actors are *reactive*, i.e. they only respond to received messages. An actor's behaviour is deterministic; the response to a message is uniquely determined by the message contents. An actor may perform three basic actions when receiving a message: (1) create a finite number of new actors (with fresh names), (2) send a finite number of messages, (3) switch to a different behaviour. Actors exhibit a maximum amount of concurrency in that all actions performed on receiving a message are done in parallel; no order is imposed. The actor model has spawned a large family of concurrent object-oriented languages [148, 75, 119, 120]. It also prompted the development of the *active object* paradigm. Active objects are very similar to actors but they are *proactive*, i.e. they have an internal schedule of actions. This schedule is expressed in the form of a *body*: a special feature (usually an infinite loop) which specifies the object's own behaviour and the order in which requests from other active objects should be processed.

America [5, 7, 6] introduces the POOL family of concurrent languages based on active objects. The problems encountered during the design of POOL-T [5] and POOL-I [7] are typical of all languages based on active objects. POOL-I is a strongly-typed language with genericity and dynamic binding. The concepts of inheritance and subtyping are clearly separated. Subtype relationship is based on a contravariant rule for feature parameters and a covariant rule for feature result. POOL-I does not have a built-in support for Design by Contract; all behavioural properties of types are expressed by property identifiers. Since it is impossible to check them automatically, the responsibility to respect the specification is on the programmer's side. This is a weak point of POOL-I. Additionally, there are no rules for subtyping and inheritance of the body of an active object. In the active object approach, the body is an essential feature determining the behaviour of objects. In general, the type of an object depends on its body. But since a formal description of this dependence would be intractable for a compiler, the programmer has the responsibility to implement the behaviour according to the specified properties. Furthermore, there is no natural and fully automatic way to make an inherited body work correctly with newly introduced features, or to combine several bodies into a new one. Therefore, in POOL-I bodies are not inherited; every class must be given a body of its own. The author claims that, since most classes only have the default body (one that accepts calls as they arrive), this does not lead to much extra work. In cases where a non-standard body is necessary, "it is well worth the trouble of paying extra attention to it". The interaction between inheritance and parallelism is limited to the obligation for the programmer to provide a body. Because subtyping and inher-

itance are not combined, the new body does not need to be compatible with the bodies of the ancestor classes [7]. The object-oriented aspects of POOL-I are further developed in [6]. The paper focuses on subtyping and genericity (including constrained genericity). America claims

> If we add a new method to a carefully designed set of variables and methods, it is quite possible that the new method invalidates an invariant on which the functioning of the old methods was based. In this way the old methods may start to behave very differently, so that we have code sharing, but no specialisation in behaviour. On the other hand, it is well known that it is often possible to obtain the same functionality by very different representations.

This comment certainly applies to languages with no support for Design by Contract. Design by Contract imposes a set of restrictions on feature redefinition to prevent inheritance anomalies mentioned in [6]. An interesting observation by the author of POOL-I is that the best way to specify formally the type of an object is "not to reason about the sequence of messages that are sent to this object, but to introduce an abstract state for each object: a mathematical entity that represents the object at a specific point during its life. Then every method can be specified by expressing its effect on that abstract state by pre- and postconditions." A similar reasoning is applied in SCOOP, although abstract states are expressed in terms of boolean predicates rather that using explicit names. Despite a limited support for inheritance, POOL-I is a major improvement on POOL-T where inheritance was completely prohibited in order to avoid the clashes with concurrency specifications.

Conflicts between synchronisation and inheritance have been tackled by many other researchers [31, 74, 117, 139]. In most O-O concurrency models they are caused by a high interdependence between the attributes of a class and the coordination constraints of different routines. Concurrency coordination and functional code are usually interwoven; as a result, features cannot be redefined in subclasses without affecting other features. Very often, the affected features must be redefined in the descendant and the ancestor, which degrades the maintainability and prevents the reuse of code. Even if the coordination code is isolated from the functional code, it is sometimes necessary to redefine it completely for all inherited features instead of having local extensions of its parts. These and other difficulties in combining inheritance with concurrency are referred to as *inheritance anomaly*; the seminal paper by Matsuoka and Yonezawa [90] formalises this notion and proposes a taxonomy of anomalies. Thanks to the new feature redefinition rules and a generalised semantics of contracts, SCOOP is immune to most anomalies (see section 10.5).

3.2 Multithreading

Multithreading with shared memory has established itself as a common model for concurrent programming. Its successful adoption by the software industry is largely due to the popularity and widespread use of programming languages such as Java, C++, and C♯. A thread represents an activity that may be run in parallel with other activities. Threads communicate by reading and writing shared memory locations. Multithreading is attractive because it allows fine-grained parallelism which often results in an increased performance of computations. It is commonly used for modelling and structuring individual tasks within a software system, although it is not well-suited for that role; the continuous discussions about how multithreading breaks the Java Memory Model [125, 89] are typical symptoms of this situation.

Java and its base libraries use locks for mutual exclusion and communication between threads [79]. Locking is fine-grained: locks protect single objects; a lock is implicitly associated with each object at its creation. Methods may be qualified as *synchronised*, in which case their invocation is mutually exclusive with other synchronised methods of the same object. If a method is not synchronised, it may be executed without mutual exclusion. This also applies to attributes which cannot be declared as synchronised. As a result, several threads may be active within one object at the same time, reading and updating its state; this leads to potential synchronisation defects.

The explicit specification and control of low-level parallelism creates new sources of programming errors: data races, atomicity violations, and deadlocks. Compared to the multithreading models, SCOOP shields programmers from the first two sources of errors. Locking is applied at the level of processors, i.e. a lock protects the whole object structure handled by a given processor. This, together with the sequential nature of processors, guarantees that SCOOP programs are data-race-free by construction. Similarly, a sequence of accesses reading or updating the state of an object structure, placed within the body of a routine that locks the processor handling the object structure, is guaranteed to execute atomically, thus avoiding unintended interference with operations of other processors. The price for safety is the increased granularity of parallelism: a routine body represents the smallest critical section; as a result, it is impossible to parallelise a computation beyond the level of a single feature call. Nevertheless, this drawback is more than compensated by the advantages in terms of safety, modularity, and convenience.

Prevention and detection of data races in multithreaded programs has been a rich research topic over the past decade. Most approaches target existing programs, typically written in Java or C++; very few propose a methodology or at least a set of rules to guide the *construction* of race-free programs. Race conditions can be avoided by careful programming discipline: protecting each data structure with a lock and acquiring that lock before manipulating the data structure. As observed by Flanagan [56], current programming tools provide little support for this discipline. It is easy to write a program that, by mistake, neglects to perform certain crucial synchronisation operations. Synchronisation errors cannot be detected by traditional compile-time checks. Furthermore, the resulting race conditions are scheduler-dependent, hence they are difficult to catch by testing.

Many run-time detection algorithms for data races have been proposed. Their key advantage is the precision of results: they report few or no false positives; this usually comes at the cost of a high run-time overhead. Eraser [130] detects data races in unannotated ANSI C programs. Working with unannotated code is a big advantage when dealing with legacy systems, e.g. for parallel scientific computations. Such tools report data races observed in a single execution; only a subset of all potential thread interleavings is considered. Therefore, certain races are not reported. Choi et al. [42] propose a methodology for Java programs that combines static data race analysis, code instrumentation, and run-time detection. They achieve good precision and efficiency. Von Praun and Gross [144] propose another approach to run-time race detection in Java programs. A conflict checker, implemented in an ahead-of-time compiler, tracks access information at the level of objects rather than individual variables. This coarser granularity optimises the analysis by restricting dynamic checks to objects identified by escape analysis as potentially shared. The algorithm largely improves on Eraser's performance by using pointer escape analysis to filter out statements that do not lead to data races. However, the coarse granularity of detection leads to the reporting of many spurious data races, e.g. when two methods

are called concurrently on an object (because every method call is viewed as a write operation). In his PhD thesis [143], von Praun further develops the approach and proposes an analysis technique that operates on an abstract model of threads and data, and simulates the execution of a parallel program on these abstract domains. This symbolic execution provides a general platform to analyse properties of concurrent programs. The approach is geared towards the detection of data races, atomicity violations of routine bodies, and deadlocks. Although it is limited by aliasing and the resulting difficulty to disambiguate dynamically allocated objects and locks, the achieved approximation yields useful practical results; overreporting may occur, however at a rate that is amenable to manual inspection. The approach is not sound; true defects may be overlooked but underreporting is limited to cases rarely occurring in practice. Two alternative software mechanisms are also developed to assess concurrency and locking at run time: an object race detection algorithm checks if every access to shared objects follows a locking discipline; an object consistency check guarantees that threads' accesses to individual objects are serialisable. Both mechanisms are implemented as a sparse program instrumentation.

Currently, a shift from run-time to compile-time approaches may be observed. Programmers become more aware of the need to apply systematic construction methods to multithreaded programs, and they are more likely to accept the burden of annotating their software to enable static analysis. Several tools, e.g. Warlock [133], use such annotations to detect potential data races. ESC/Java [47, 82] uses an underlying theorem prover to check the absence of data races and deadlocks. More and more proposals are based on type systems [15, 56, 59, 58]. The interest in using Design by Contract in a concurrent context is also growing. Rodriguez et al. [128] present an extension of JML which covers mutithreading. New constructs introduced to JML rely on *method atomicity* which supports reasoning about non-interference properties of features. Hoare-style reasoning about concurrent programs is supported; annotated programs may be verified using model checking. The authors observe that "existing specification and checking tools for multithreaded software typically focus on atomicity properties such as mutual exclusion but they ignore strong functional properties and complex invariants". This is due to the difficulty of dealing with thread interference. Standard Hoare-style reasoning using pre- and postconditions does not apply because other threads may invalidate the assumptions made by the thread executing a given feature. When reasoning about semantics of features, it would be necessary to consider all possible interleavings to account for the interference between threads. This is much more expensive than reasoning in terms of pre- and postconditions. On top of that, such approach is not modular, i.e. it is impossible to analyse classes individually. A similar observation provided part of the impetus for our work: we wanted to find a generalised rule for reasoning about the correctness of feature calls, that would be applicable in the concurrent and the sequential contexts and boil down to the standard sequential rule when no concurrency is involved (see chapter 8 for details). Two kinds of interference between threads are tackled in [128]. *Internal interference* arises when another thread modifies the data that the current thread relies upon. For example, the current thread evaluates the precondition of feature f and then starts to execute its body; another thread modifies the state of some objects in such a way that the precondition of f does not hold anymore. The current thread still works on the assumption that the precondition holds; as a result, it may fail to satisfy the postcondition of f. *External interference* arises when another thread makes observable state updates between a feature call and the feature entry, i.e. the point at which the feature body is executed by the supplier object. Similar interference may occur between the moment when the execution of a body terminates and the moment when the client object resumes the execution of its feature.

Internal interference makes it impossible to use Hoare rules for reasoning about correctness of feature bodies; external interference invalidates reasoning about feature calls. The approach assumes a sequential consistent memory model which is stronger than the actual Java memory model known for certain low-level data races [125]. Therefore, only race-free programs are considered; race-freedom is assumed to be dealt with using other techniques, e.g. PRFJ [56]. The verification methodology has been implemented using the Bogor model checker [127]. The tool is able to check the atomicity of features that exhibit a behaviour compatible with Lipton's theory [83]. Some other types of atomicity can be checked by Atomizer [57]. Nevertheless, atomicity of features that use more complex synchronisation patterns cannot be verified by these tools.

Flanagan and Freund [56] present a static race detection analysis for multithreaded programs written in ConcurrentJava which is a small subset of Java with an additional **fork** construct. The analysis is based on a formal type system capable of capturing many common synchronisation patterns, such as classes with internal synchronisation, client-side synchronisation, and thread-local objects. A class definition may contain a sequence of formal parameters or *ghost variables* that decorate types of entities. Ghost variables are used by the type checker to verify that the program is data-race free. Ghost variables do not affect the run-time behaviour of programs; they may only appear in type annotations and not in regular code. They are similar to *processor tags* in our type system (see section 6.2.2); however, processor tags offer much more flexibility, in particular in the context of (multiple) inheritance and polymorphism. ConcurrentJava provides mechanisms for escaping the type system in places where it proves too restrictive, or where a particular data race is considered benign.

Boyapati and Rinard [29] propose a type system for multithreaded programs. Their type system makes it possible to associate with each object an appropriate synchronisation mechanism ensuring the absence of data races. The synchronisation mechanism is specified through the type of variables that reference the object. In general, an object can be safely accessed by a thread if that thread has acquired a lock associated with the object. A thread is also allowed to access an object without acquiring a lock if either (1) the object is immutable (read-only), (2) the variable is a unique reference to the object, or (3) the object is thread-local, i.e. only accessible to one thread. The type-checker uses type specifications to verify that all accesses to objects comply with the declared synchronisation policies. The supported language, Parametrised Race-Free Java (PRFJ), is an extension of ConcurrentJava. The syntax is enriched to enable specification of object ownership and locking requirements. In PRFJ, every object has an owner: the object itself, another object, or **thisThread**. If an object is owned by **thisThread** (directly or indirectly), it is local to the corresponding thread and it cannot be accessed by other threads. The ownership is fixed, i.e. objects cannot change their owners. The ownership relation forms a forest of rooted trees; self-loops in roots are allowed. To gain an exclusive access to an object, a thread has to acquire the lock on the root of the ownership tree that contains the object. The locking requirements of a method can be specified using the **requires** clause. A method may require callers to hold one or more locks before calling it. To specify ownership, classes may be parametrised with one or more owner parameters. The first one always denotes the owner of the current object (**this**). Each owner parameter must be instantiated when the given class is used to declare the type of an entity. Possible instantiations for an owner parameter are: **self** (in which case the object is owned by itself), **thisThread**, a final field, a final variable, or a formal owner parameter of the enclosing class. The latter supports propagating the ownership information, so that a whole data structure may be guarded by a single lock. Since fields and

variables used to instantiate owner parameters must be final, i.e. they cannot be modified after creation and initialisation, the owner of an object does not change over time. Requiring that these fields and variables be already initialised gives an additional guarantee that every new object is owned by an already existing object, by itself, or by **thisThread**. These requirements are very similar to the requirements put on *qualified processor tags* in our framework (see section 6.2.2). PRFJ handles read-only objects and unique pointers using additional type annotations *!w* and *!e*. Types of formal arguments and local variables may be augmented with *!e* to mark them as non-escaping. Unique pointers can only be assigned to non-escaping entities. Similarly, a read-only object may be passed as argument to a feature call only if the feature declares the corresponding formal argument as read-only (*!w*). This solution is heavy and it is not clear how well polymorphism is supported. The *expanded types* mechanism provides a simpler solution for unique pointers (see section 6.10). To minimise memory usage and run-time overhead, PRFJ does not preserve ownership-related information at run time. As a result, downcasts cannot be properly verified. In our framework, the locality information is preserved at run time; every object (of a reference type) carries an implicit reference to its handler. Therefore, we are able to handle downcasts using the refined *object test* mechanism (see section 6.7).

Boyapati et al. [28] enrich PRFJ with type annotations for deadlock-prevention. Programmers have to partition all locks into equivalence classes (called *lock levels*) and specify a partial order between these classes. The type checker makes sure that there is no cycle in the partial order relation, and that every thread acquires locks in a descending order of lock levels. Additionally, locks within a single lock level may be ordered using a recursive tree-based data structure. For example, one can specify that nodes in a tree have to be locked in a tree order (first parent node, then children nodes). The type system also allows modifications of the partial order at run time, provided that they do not introduce cycles. Except for that last feature, the type system is very restrictive. In particular, the requirement that all potential locks be partitioned into a constant number of lock levels makes it very difficult to write reusable code; when writing a library class, it is usually impossible to anticipate all the potential uses of the class. A partial solution of this problem is provided in the form of classes parametrised with lock levels. The use of condition variables (with *wait* and *notify*) is restricted: a thread may only wait on a condition variable *e* if it holds a lock on *e* and no other locks. This solves the *nested monitor problem*, i.e. a situation when a thread is suspended without releasing all its locks. Since the thread releases its lock on *e* when it suspends itself, and no other locks are held by the suspended thread, deadlocks cannot happen. Nevertheless, SCOOP-style combination of condition synchronisation and atomic locking of several objects cannot be implemented.

Jacobs et al. [69] introduce a methodology for protecting object structures from inconsistencies due to race conditions. The proposed approach is an extension of the Boogie method [19] which enables modular verification of object invariants that depend on mutable state of other objects. The dynamic ownership model is extended to allow threads as owners. The semantics of **pack** and **unpack** statements is refined to permit ownership transfer from a thread to an object structure and vice-versa. Moreover, new statements **acquire** and **release** permit the acquisition and release of ownership on consistent object structures. The proof methodology is based on sequential reasoning about field accesses. To support sequential reasoning, interference between threads must be controlled, i.e. a thread must have exclusive access to all the relevant fields during the execution of a program fragment under scrutiny. A thread is required to own (transitively) an object whenever it reads or writes one of its fields. Owner uniqueness imposed by the Boogie method guarantees mutual exclusion. The rules for own-

ership are extended as follows. A newly created object is mutable and owned by the creating thread. A thread may attempt to acquire ownership on an object; the **acquire** operation blocks until the object is free. When it succeeds, the object may be assumed to be in a consistent state (i.e. its invariant holds). A thread can relinquish ownership by executing **release**. An **unpack** operation on a consistent aggregate object transfers the ownership on the representation objects from the aggregate object to the executing thread. This ownership can be transferred back to the aggregate object by performing a **pack** operation. The methodology supports client-side and supplier-side synchronisation. Client-side synchronisation avoids the problem of internal interference, thus enabling reasoning with pre- and postconditions. Assertions on methods accessed following this synchronisation style are similar to SCOOP's *controlled assertions* (see section 8.2). Supplier-side synchronisation with **acquire** and **release** is similar to SCOOP's argument-based locking (see section 6.1.3) but limited to a single object at a time (atomic locking is not supported).

3.3 Concurrency in Eiffel

Eiffel//

Denis Caromel proposes a concurrent extension of Eiffel that relies on the notion of *process*: an active object that executes a prescribed behaviour (*body*) [38]. Processes are instances of a predefined class *PROCESS*. Not all objects are processes; *passive objects*, for example, have no associated behaviour. The body of a process is implemented by the feature *live*. By default, the body just loops forever, accepting all incoming requests and servicing them in a FIFO order. It may be redefined in descendants to implement a particular behaviour. The body of an active object may access directly its request queue.

In Eiffel//, feature calls on a different active object are always asynchronous. If a feature returns a result, lazy synchronisation takes place, i.e. the call returns immediately even if its result has not been evaluated yet; a client object only needs to wait if it tries to access the value of the result. This approach, known as *wait by necessity*, enables more asynchrony that the mechanism used in SCOOP (and also referred to as *wait by necessity*). In SCOOP, the client object always waits for the result of a query call, even if it only uses that result much later. An optimisation similar to Caromel's original mechanism was described by Meyer [94] but it has never been implemented in any SCOOP system. Wait by necessity is related to the concept of *futures* found in languages such as ABCL/1 [148], ABCL/f [138], and ConcurrentSmalltalk [147]. Futures allow more asynchrony but they require special syntactic constructs, as opposed to the wait by necessity which is fully automatic and transparent to the programmer.

Eiffel// prohibits sharing of data between processes. A non-process (passive) object is only accessible to one process object; the passive object belongs the context of that process object. If a feature call is between two process objects, actual arguments are passed by deep copy. Only the arguments representing process objects are passed by reference. This solution eliminates concurrent accesses to passive objects. In our framework, where all objects may be seen as passive, a different technique is used: every object belongs to its handling processor and no sharing of objects between processors is permitted, even though feature arguments are passed by reference (except for expanded types, for which we propose an adequate solution in chapter 9).

Rather than just introducing a new language, Caromel describes a full methodology for the

development of concurrent object-oriented programs. The methodology comprises four basic steps: (1) sequential design and implementation, (2) process identification, (3) process programming, and (4) adaptation to constraints. The first step corresponds to the design of a purely sequential object-oriented system where concurrency is not taken into account. Steps 2 and 3 correspond to the parallelisation of that sequential system. The last step is the "fine-tuning" of the system according to the particular requirements such as real-time constraints, the underlying execution platform, etc. An interesting feature of Eiffel// which directly supports that design methodology is the conformance between non-process and process types. An entity declared as x: *REGULAR_CLASS* might become attached at run time to a process object, provided that this object's base class inherits from *REGULAR_CLASS*. As a result, an implicit change of call semantics might occur — communication with the (polymorphically) active object becomes asynchronous, and transmission of feature parameters is done by copy. Caromel claims that, although it changes the local semantics of calls, "this is often a desired change when parallelising and, in many cases, does not affect the global semantics of the application" [14]. Furthermore, he states that such use of polymorphism brings real benefits regarding code reusability and design methodology for concurrent systems. He summarises his approach as follows: "Think of a concurrent system as a sequential one — writing or reusing self-contained classes."

The desirability of implicit parallelisation is debatable; the global semantics of an application is changed in most cases and the clients lose the possibility to reason sequentially about feature calls. In our approach, the conformance relation on types allows for subtyping of a separate type by a non-separate one. Although there is no one-to-one correspondence between separate objects and Eiffel// processes on one hand, and non-separate objects and Eiffel// passive objects on the other hand, our use of polymorphism goes in the opposite direction to Caromel's. So does our philosophy: "Think of a sequential system as a particular case of a concurrent system."

CEiffel

Löhr [85, 86] proposes another concurrent extension of Eiffel. There are five kinds of annotations:

- concurrency: −−||−−
- compatibility: −−||...−−
- delay: −−@−−
- autonomy: −−v−−
- asynchrony: −−>−−

In CEiffel, classes may be sequential (atomic), concurrent, or semiconcurrent. A class that does not allow any overlapping of feature calls is called *atomic*. Strict atomicity can be relaxed to support overlapping of compatible features. A class that permits arbitrary overlapping is called *concurrent*. A class that permits certain overlappings but prohibits others is called *semiconcurrent*.

In a concurrent class, features may be declared as *compatible* with respect to other features. For example, if the class X is declared as **class** X `--||--`, then its feature r declared as r (...) : T **is** `--|| s, t --` is compatible with the features s and t. If the feature r is declared as r (...) : T **is** `--||--`, without an explicit list of names, then r is compatible with all the features of X, including r itself. Compatibility is a symmetric relation, i.e. it is not necessary to list r in the compatibility clauses of s and t. Compatibility is usually not transitive and it does not have to be reflexive. Compatibility annotations only matter for *invoked* features, i.e. features called on a *controlled* object. They do not apply to local (sequential) calls. Mutual exclusion of incompatible operations is ensured through a locking mechanism: a request that is incompatible with ongoing operations will be held on wait until they terminate.

In CEiffel, preconditions on non-controlled (sequential) objects keep the usual correctness semantics. Preconditions on controlled objects are split into a *checker* and a *guard*. A violated checker raises an exception; a violated guard causes the request to wait until the guard is satisfied (and no incompatible operation is being executed). The *delay annotation* `--@--` separates the two parts of a precondition, e.g.

```
enqueue (item: T) is
    require
        item /= Void      -- checker, correctness semantics
        --@--
        length < size     -- guard, wait semantics
    do
        ...
    end
```

Delay annotations are simply ignored in a sequential setting and guards are treated as usual preconditions. An interesting feature of CEiffel is the possibility of inserting a delay annotation in a postcondition; such postconditions cause a delay until they are satisfied. This seems to go in a similar direction as our framework, although the client is forced to wait; see section 8.1.2 for a discussion of postconditions in SCOOP.

Routines may also be annotated with the *autonomy* tag `-->--`. A class that has at least one autonomous routine is also called autonomous; so are its instances. When an autonomous object is created, all its autonomous routines are implicitly invoked. When an autonomous routine terminates, an immediate implicit invocation of the same routine follows. Preconditions of autonomous routines may contain guards but no checkers are allowed (since there are no clients to satisfy those). Naturally, autonomous routines take no arguments. Explicit invocations of autonomous routines are allowed, although such routines are usually not exported.

An exported routine decorated with a tag `--v--` is called *asynchronous*. A class that has at least one asynchronous routine is also called asynchronous; so are all its instances. (Note that a class may be both asynchronous and autonomous.) A client calling an asynchronous routine continues the execution right after the evaluation of the checker, without waiting for the supplier to process the request. The asynchrony annotation does not apply to local calls. Invocations of asynchronous functions use wait by necessity for synchronising with the supplier object.

The *control annotation* `--!--` attached to the type declaration of an entity indicates that the object represented by the entity is *controlled*.

Since concurrency annotations are sprinkled all over the client and the supplier code, it is difficult to apply modular reasoning to CEiffel programs. Control annotations embedded in

clients make it impossible to reason about the semantics of a supplier without inspecting the client code. Conversely, it is necessary to analyse the code of suppliers to understand the semantics of a client because asynchrony annotations used in the suppliers influence the client. Furthermore, the atomicity of call composition involving asynchronous invocations is not guaranteed.

Concurrency class

Karaorman and Bruno [76] propose a concurrency library for Eiffel. Concurrency is achieved through parallel execution and interplay of several active objects. Objects are active if their base class inherits from a special class *CONCURRENCY*. The body of an active object is implemented by the feature *scheduler*. The *CONCURRENCY* class implements asynchronous, non-blocking invocation mechanism. Clients issue requests using *remote_invoke*. This feature sends a request to a supplier and returns a request id without waiting. The request id may be used later on to claim the result of the invoked feature. This is done by calling *claim_result*, which blocks until a result is available. A non-blocking test for result availability, *result_ready*, is also provided. Explicit invocation of *claim_result* may be eliminated through the use of the *FUTURE* type. Synchronisation becomes then similar to wait by necessity in Eiffel//.

Scheduling of requests depends on the body of a given active object. The body has an unrestricted access to the request queue (through the feature *request_queue*); it can retrieve names and parameters of all requests stored in the queue. This enables fine tuning of scheduling policies. For example, the body may choose to wait until a request of a certain type is in the queue. It is also possible to check for the presence of any requests (this is a non-blocking operation) or accept the next available request (this is a blocking operation if the queue is empty). Sending the result of a request to the client is performed explicitly using the feature *send_result*.

Distributed Eiffel

Distributed Eiffel [62], designed as language support for the distributed operating system Clouds [44], offers the possibility to qualify features as *readers* or *writers*, using the corresponding annotations *ACCESSES* and *MODIFIES*. If a reader feature is called, a *read lock* is acquired on the target object before the feature is executed. Similarly, calling a writer feature requires a *write lock*. If a feature is not qualified as reader or writer, no locks are necessary. Therefore, Distributed Eiffel supports unrestricted intra-object concurrency. An additional *WHEN* construct is used for declaring feature guards; guards are specified as predicates on the state of the supplier object. Three parameter-passing modes are used: *copy* (standard pass-by-value), *copy out - copy in* (pass-by-value in both directions), and *PIN* (pass-by-reference). The passing mode is specified on the client side.

Distributed Eiffel does not provide any special mechanism for resynchronising clients and suppliers; programmers have to implement the resynchronisation explicitly.

CSS/CEE

In her PhD dissertation [70], Ghinwa Jalloul proposes CSS: a conceptual model for object-oriented concurrency where objects are structured into sequential subsystems. A concurrency layer consisting of concurrent objects (one per subsystem) is responsible for communication, synchronisation, internal concurrency control, remote reference passing, and resolution of remote dynamic binding. Although the general structure of a CSS system resembles the SCOOP processor model, there are some important differences:

- In CSS, concurrent objects may be referenced by any object in the system but sequential objects may only be referenced by other objects within the same subsystem. SCOOP has no "concurrent" and "sequential" objects; every objects can be referenced by any other object in the system.
- CSS supports intra-object concurrency based on the reader–writer scheme. Interestingly, calls issued by a concurrent object to itself are also handled asynchronously. SCOOP processors are sequential; no intra-object concurrency is permitted.

Jalloul also introduces CEE: a concurrent extension of Eiffel which supports the CSS model. Atomic locking and condition synchronisation on the client side are achieved through an explicit CCR-like construct **holdif**. Supplier-side synchronisation is specified as part of routine contracts but an explicit notation **guard** *exp* is used to distinguish guards from regular preconditions. CSS allows keeping or strengthening the guards in redefined features. This is one of the major differences with respect to our approach where synchronisation conditions specified as precondition clauses may only be kept or weakened in descendants.

CSS and CEE permit more asynchrony than SCOOP but they do not support contract-based reasoning about concurrent programs.

4

Original SCOOP_97 model

IN the quest for a convenient method of concurrent programming, we have soon realised that the original SCOOP_97 model [94] is a good starting point and an important first step towards providing a coherent framework for concurrency. It relies on the implicit concurrency present in the feature call mechanism; it also makes smart use of argument passing, preconditions, and the command-query distinction to provide the necessary synchronisation. At first, the model seems to "hijack" these mechanisms; a closer inspection, however, reveals a deeper reason behind the apparent semantic overloading. To grasp this deeper reason and uncover the precise semantics of O-O mechanisms — thus opening the way to a better understanding of object technology in general — it is necessary to analyse the model in detail, find its inconsistencies and limitations, clarify its rules, and extend it to cover the full spectrum of object-oriented techniques.

SCOOP_97 has undergone several reviews since its initial publication [91]; therefore, we start with a short historical overview of its development. A detailed presentation of the model follows; it is based on the description found in [94]. We wrap up with a discussion of related work by other authors. The critique in chapter 5 points out the inconsistencies and limitations of the model; it serves as a roadmap for the development of the current SCOOP framework presented in chapters 6 – 10.

4.1 Development

SCOOP_90: TOOLS

The first informal description can be found in a TOOLS'90 article by Bertrand Meyer [91]. The article suggests the use of implicit concurrency present in the feature call mechanism, and introduces the basic computational model: processors, separate objects, and wait-conditions. The proposal was originally known as PEiffel.

SCOOP_93: CACM

The 1993 CACM article "Systematic Concurrent Object-Oriented Programming" [93] presents the model as a minimal extension of sequential Eiffel to support concurrency and distribution. To address such requirements of concurrent programming as mutual exclusion and condition synchronisation, new semantics is given to well-known constructs, e.g. preconditions, where the standard sequential semantics cannot be applied.

SCOOP_97: OOSC2

Meyer's OOSC2 book [94] (chapter 30), published in 1997, gives a detailed description of the model; it is essentially SCOOP_93 enriched with two advanced mechanisms: *duels* and *CCF*. The exception-based duel mechanism enables priority scheduling: an impatient client may request immediate service which results in an exception raised in the current holder (if the request is granted) or in the requesting client (if the request is refused). The CCF mechanism (*Concurrency Control File*) maps processors to physical resources: programmers can specify on which physical resources (machines, CPUs, processes) a newly created processor will be placed.

The book gives the rationale for the application of O-O mechanisms in a concurrent context, discusses thoroughly the practical use of the model, and presents several programming examples. Since we take the description from OOSC2 to be the starting point of our study, all discussions concerning the original model refer to SCOOP_97; so does the rest of this chapter.

4.2 SCOOP_97 in detail

4.2.1 Processors and separate objects

In the O-O world, to perform a computation is to use certain *processors* to apply certain *actions* to certain *objects* [94]. Actions are represented by features. A feature call $x.f\ (a)$ has the following semantics: the client object applies feature f on the supplier object attached to x, with argument a. So, it mentions actions and objects; what about processors? In a sequential setting, there is only one processor, i.e. one thread of control; therefore, it remains implicit. In a concurrent context, there are two or more processors; this is the essential distinction between concurrency and sequentiality. SCOOP_97 achieves concurrency through the interplay of several processors.

Definition 4.2.1 (Processor) *A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects.*

Every object has a *handler*: a processor in charge of executing feature calls on that object; no processor other than the handler is allowed to manipulate the object. Several objects may have the same handler; the mapping between an object and its handler does not change over time. Objects handled by different processors are called *separate*; objects handled by the same processor are *non-separate*. References between separate objects are called *separate references*; such references cross the boundaries of processors. The semantics of a feature call depends on whether the supplier object is separate from the client object. Consider a situation where the client object *o1* performs calls

$x.f$
next_call

with entity x attached to the supplier object *o2*. If the supplier object is non-separate from the client object, like in figure 4.1, then the call $x.f$ is synchronous, just like in a sequential context; *o1* must wait for *o2* to finish the execution of f because there is only one processor to handle both objects. If the objects are separate, like in figure 4.2, the call becomes asynchronous, i.e.

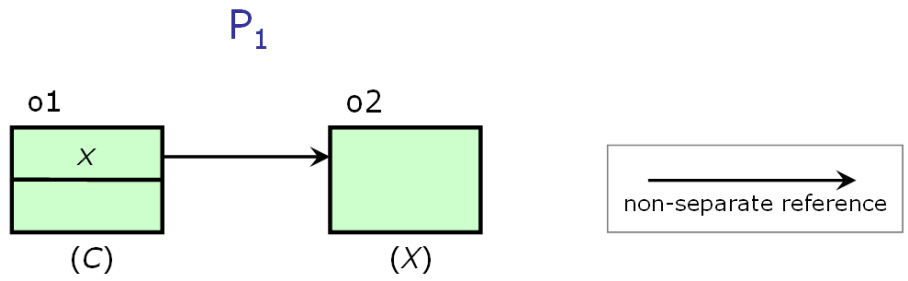


Figure 4.1: Non-separate objects

the computation on *o1* can move ahead to *next_call* without waiting for $x.f$ to terminate; this is because two different processors — P_1 and P_2 — handle the involved objects.

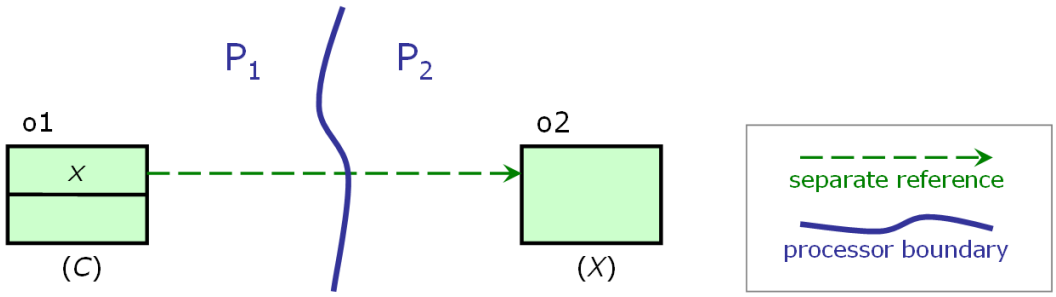


Figure 4.2: Separate objects

A processor, together with the object structure it handles, forms a sequential system. (We use the term *processor* to refer to such a system; other authors [43, 60, 34] use the term *subsystem*, while *processor* only denotes the associated thread of control.) Therefore, every concurrent system may be seen as a collection of interacting sequential systems; conversely, a sequential system may be seen as a particular case of a concurrent system (with only one processor). Figure 4.3 shows a typical SCOOP_97 system.

Viewed by the software, a processor is an abstract concept; the same concurrent application may be executed on very different architectures without any change to its source text. A processor does not have to be associated with a physical CPU; it may also be implemented by a process of the operating system, or a thread in a multithreading environment [111]. In the .NET framework, processors can be mapped to application domains [112].

4.2.2 Separate entities, classes, and calls

Since the effect of a call depends on whether the client and the supplier objects are handled by the same processor or by different ones, the software text must distinguish unambiguously between these two cases. For declarations of variables or functions, normally appearing as

x: X

a new form is now possible:

x: **separate** X

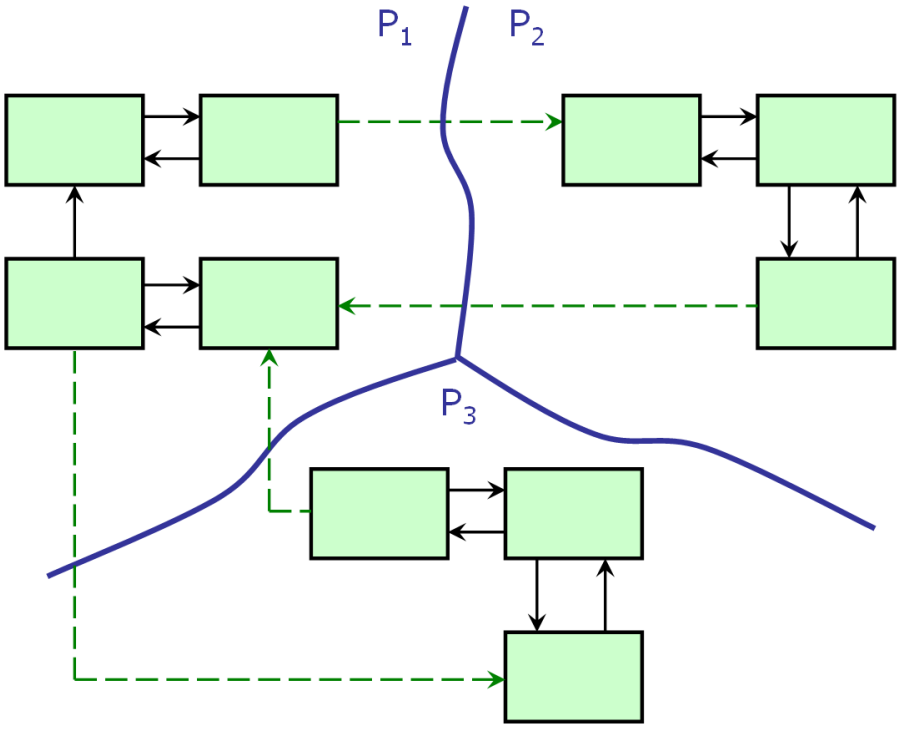


Figure 4.3: Concurrent system composed of several processors

Keyword **separate** indicates that *x* is a *separate entity*, i.e. it may become attached to objects handled by a different processor. The **separate** declaration does not specify which processor to use for handling the object. What matters is that the processor is different from the processor handling the current object. Rather than making an individual entity separate, it is also possible to declare a class as separate[1], as in

separate class *SOME_CLASS*
...
end

Any entity of the corresponding type, e.g. *y*: *SOME_CLASS*, is implicitly separate. *SOME_CLASS* is called a *separate class* and all its instances are separate from all other objects.

The value of a *separate entity* is a *separate reference*; if not void, it is attached to an object handled by another processor — a *separate object*. If *x* is a separate entity, any creation instruction **create** *x.make* (...) will associate a fresh processor with the newly created object. Calls on separate entities are asynchronous; they are referred to as *separate calls*.

4.2.3 Synchronisation

SCOOP_97 caters for the synchronisation and communication needs of concurrent programming, such as mutual exclusion and condition synchronisation, by relying on preconditions and argument passing. Since each object may only be manipulated by its handler, there is no object

[1]The **separate** annotation in classes cannot be combined with **expanded** or **deferred**.

sharing between different threads of execution, i.e. no shared memory. The sequential nature of processors and the resulting absence of intra-object concurrency mean that programs are data-race-free by construction. Nevertheless, an additional mechanism is necessary to eliminate *atomicity violations*, i.e. illegal interleaving of calls from different clients, and *high-level data races*, i.e. unordered accesses to data structures [143, 13].

Mutual exclusion, atomicity

Two basic rules ensure the mutual exclusion and atomicity properties:

Definition 4.2.2 (Separate Call rule) *The target of a separate call must be a formal argument of the routine in which the call appears.*

Definition 4.2.3 ((Partial) Wait rule) *A routine call with separate arguments will execute when all processors handling these arguments are available to the client; the client will hold the processors for the duration of the routine.*

The call *my_x.f* in figure 4.4 is invalid because its target is not a formal argument. On the other hand, calls *x.f* and *x.g* are valid. The client calling *r (my_x)* blocks until the processor handling *my_x* is available; the client will retain exclusive control over that processor (through locking) until the execution of *r* terminates. Therefore, all calls on *x* within *r* are executed in mutual exclusion with respect to other clients; additionally, no other client is allowed to access *x*'s processor between the two calls. Rules 4.2.2 and 4.2.3 enforce the basic synchronisation policy

```
r (x: separate X)
    do
        ...
        x.f
        ...
        x.g
        ...
    end

my_x: separate X
...
r (my_x)
my_x.f          -- Invalid!
```

Figure 4.4: Mutual exclusion

that guarantees atomicity of features involving separate calls; a routine body represents a critical section with respect to to its separate formal arguments. This prevents a common mistake in concurrent programming that consists in assuming that, when making two successive calls on a separate object, e.g.

```
my_stack.push (some_value)
...
x := my_stack.top      -- x = some_value?
```

nothing may happen to the object represented by *my_stack* between the two calls. In the example above, we would expect that the object assigned to *x* is indeed the object denoted by *some_value* that we just pushed on *my_stack*. Unfortunately, such "sequential" thinking does not apply in a concurrent setting because other clients may interfere with *my_stack* between the two calls. This is a typical atomicity violation we want to avoid.

If a routine takes several separate arguments, all of them are locked *atomically* before the routine is executed. For example, the execution of

```
eat ( left_fork , right_fork )
```

blocks until both arguments have been locked. There is no limit on the number of formal arguments in routines, hence no limit on the number of locks that may be acquired atomically.

Condition synchronisation

In sequential programs, a precondition is a correctness criterion that the client object must fulfil before calling a given routine on the supplier object. If the precondition is not met, the client has broken the contract; for example, it has tried to store a value into a full buffer. Since the execution is sequential, the state of the buffer cannot change — no other client may access the buffer in the meantime. In a concurrent context, this does not apply any more. Consider the feature *store* in figure 4.5. Its precondition requires that *buffer* not be full and *i* be positive. A non-satisfied precondition clause **not** *buffer . is_full* which involves a separate call does not break the contract; instead, it forces the client to wait until it is satisfied. (As a result of other clients' activity, the state of *buffer* may eventually change in a way that satisfies the precondition.) On the other hand, the precondition clause $i > 0$ preserves its correctness semantics; if violated, it raises an exception.

```
store ( buffer : separate BUFFER [INTEGER]; i: INTEGER)
        -- Store i in buffer .
    require
        not buffer . is_full
        i > 0
    do
        buffer .put ( i )
    end

my_buffer : separate BUFFER [INTEGER]
...
store ( my_buffer , 10)
```

Figure 4.5: Preconditions vs. wait conditions

The application of the standard correctness semantics to *separate preconditions*, i.e. precondition clauses that involve separate calls, would lead to the *Separate Precondition paradox*: suppliers cannot do their work without the guarantee that the precondition holds; but the clients are unable to ensure these preconditions for separate arguments. Even if the client in figure 4.5 performed a test on *my_buffer* before calling *store* , e.g.

```
if not my_buffer.is_full then
    store (my_buffer, 10)
end
```

the state of *my_buffer* might have changed between the test and the moment of the call; as a result, the precondition might be violated. Therefore, the *wait semantics* must apply to separate preconditions; correctness semantics applies to other preconditions.

The wait semantics makes it possible to define precisely the behaviour of feature calls with separate arguments, thus completing the *wait rule*.

Definition 4.2.4 (Wait rule) *A routine call with separate arguments will execute when all processors handling these arguments are available to the client and the separate preconditions are satisfied; the client will hold the processors for the duration of the routine.*

Wait by necessity

Due to the asynchronous semantics of separate calls, a client executing them is not blocked; it can proceed with the rest of its computation. Later on, however, it may need to resynchronise with the supplier, for example to retrieve some results. Rather than introducing a specific language mechanism for this purpose, SCOOP_97 relies on the *wait by necessity* principle.

Definition 4.2.5 (Wait by necessity) *If a client has started one or more calls on a certain separate object, and it executes on that object a call to a query, that call will only proceed after all the earlier ones have been completed, and any further client operations will wait for the query call to terminate.*

Figure 4.6 illustrates the concept: the client waits on the query call $x.some\_query$, whereas procedure calls $x.f$, $x.g\ (a)$, and $y.f$ do not cause waiting. SCOOP_97 applies a restricted version

```
r (x: separate X)
    do
        x.f             -- No waiting here
        x.g (a)         -- No waiting here
        y.f             -- No waiting here
        ...
        value := x.some_query    -- Wait here for result
        ...
    end
```

Figure 4.6: Wait by necessity

of wait by necessity. The original mechanism, first introduced in Eiffel// [38], permits even more asynchrony: a query call returns immediately even if its result has not been evaluated yet; a client object only needs to wait when it tries to access the value of the result. An optimisation in the spirit of the original principle is described in OOSC2 but it has never been implemented.

4.2.4 Consistency rules

Since the semantics of separate and non-separate calls is different, it is essential to guarantee that a non-separate entity, e.g. x: X (where class X is not separate), never becomes attached to a separate object. Otherwise, calls on x would be wrongly processed as synchronous calls, without respecting rules 4.2.2 and 4.2.4, i.e. the client would be able to use x without locking the corresponding processor beforehand. Entities declared as non-separate but pointing to separate objects are called *traitors*. SCOOP_97 provides four *separateness consistency rules* to eliminate traitors.

Definition 4.2.6 (Separateness consistency rule SC1) *If the source of an attachment (assignment instruction or argument passing) is separate, its target entity must be separate too.*

Rule SC1 eliminates the risk of introducing a traitor through assignment or argument passing. For example, the assignment *my_y* := *my_x* in figure 4.7 is invalid because its source is separate but its target is not. Similarly, the call r (*my_x*) is invalid because the actual argument *my_x* is separate while the corresponding formal x is not. If the call was valid, then x would become a traitor; the call $x.f$ would be executed without locking x's handler, potentially violating the atomicity of other clients' operations involving that object.

Note that there is no rule prohibiting attachments in the opposite direction — from non-separate to separate entities. Meyer [94] states: "permitting an attachment of a non-separate source to a separate target is harmless — although usually not very useful."

```
r (x: X)
    do
        x.f
    end

my_x: separate X
my_y: X
 ...
my_y := my_x      -- Invalid
r (my_x)          -- Invalid
```

Figure 4.7: Application of rule SC1

Definition 4.2.7 (Separateness consistency rule SC2) *If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.*

Rule SC2 takes care of the situation where a reference to a non-separate object is passed — as actual argument of a separate call — across the boundary of a processor. Such a reference must be seen as separate outside that boundary; to ensure this, the corresponding formal argument must be declared as separate. Figure 4.8 illustrates this scenario. The client is not allowed to use its non-separate attribute a as actual argument of $x.f$ because the feature f in class X takes a non-separate formal argument; if this was permitted, that formal argument would become a traitor. On the other hand, the call $x.g$ (a) is valid because g takes a separate formal argument.

```
-- in class C
a: A

r (x: separate X)
    do
        x.f (a) -- Invalid
        x.g (a) -- Valid
        ...
    end

-- in class X
f (a: A)
    do
        ...
    end

g (a: separate A)
    do
        ...
    end
```

Figure 4.8: Application of rule SC2

Definition 4.2.8 (Separateness consistency rule SC3) *If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate.*

A non-separate reference may also be passed across a processor's boundary as a result of a separate call to a function. Rule SC3 takes care of that problem, as illustrated in figure 4.9. The assignment to a is invalid because $x.f$ is a separate function call but a is non-separate. The assignment to b is valid because b is separate. In a sense, rule SC3 "mirrors" SC2.

The problems handled by the rules SC2 and SC3 only concern reference types; they do not apply to expanded objects because such objects are always passed by copy, so there is no danger of creating a traitor which looks like a non-separate object but "sits" on a different processor. Nevertheless, non-separate references carried by expanded objects may constitute a danger; therefore, the use of expanded types is restricted by the rule SC4.

Definition 4.2.9 (Separateness consistency rule SC4) *If an actual argument or result of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.*

Figure 4.10 illustrates the problem. Class *EA* is expanded, i.e. all its instances are expanded. Without the rule SC4, the assignment $a := x.f$ in the client's code would be valid. But then the client would be able to call $a.b.f$ which is obviously invalid because $a.b$ — although declared as non-separate — denotes a separate object; $a.b$ is a traitor. The problem stems from the fact that the source of the assignment $a := x.f$ is passed by copy; it also carries a copy of its non-separate reference b which immediately becomes a traitor when it crosses the boundary of x's

```
-- in class C
a: A
b: separate A

r (x: separate X)
    do
        a := x.f -- Invalid
        b := x.f -- Valid
    end

-- in class X
f: A do ... end
```

Figure 4.9: Application of rule SC3

processor. There would be no problem here if class *EA* did not have any non-separate attributes, i.e. if it was "fully-expanded". Therefore, SC4 only allows instances of fully-expanded classes as actual arguments or results of separate calls. Several library classes, such as *BOOLEAN*, *INTEGER*, *REAL*, and *CHARACTER* are fully-expanded; they may be used freely in separate calls.

4.2.5 Additional rules and mechanisms

Business Card principle

Rule SC2 allows a non-separate reference as actual argument of a separate call, provided that the corresponding formal argument is declared as separate. Consider the call $x.g\ (a)$ in figure 4.8. If the body of g does not perform any calls on the formal argument a but only uses it as a source of an assignment, e.g.

```
g (a: separate A)
    local
        my_a: separate A
    do
        my_a := a
    end
```

then g may simply execute without locking the client's processor — recall that a is non-separate from the client, hence handled by its processor — and the client is not hindered in its execution. If the body of g does perform some call on a, e.g.

```
g (a: separate A)
    do
        ...
        a.f
        ...
    end
```

```
expanded class EA
feature
    ...
    b: B      -- B is a reference type
    ...
end

-- in class C
a: EA         -- a is expanded

r (x: separate X)
    do
        a := x.f      -- Invalid
        a.b.f         -- a.b is a traitor
    end

-- in class X
f: EA
    do
        ...
    end
```

Figure 4.10: Application of rule SC4

then the execution of g must obey the wait rule (4.2.4); it will block until a's processor becomes available. This is likely to produce a deadlock: the client may be busy due to wait by necessity, e.g. executing a query call on x. In that case, x would wait for the client, and the client would wait for x — a classic deadlock. Therefore, SCOOP_97 introduces the *Business Card principle* which prevents such situations.

Definition 4.2.10 (Business Card principle) *If a separate call uses a non-separate actual argument of a reference type, the routine should only use the corresponding formal as source of assignments.*

Assertion Argument rule

To avert deadlock situations caused by blocking calls in preconditions, e.g.

```
a: separate A
r (x: X)
    require
        some_property (a)    -- Potentially blocking due to the wait rule
    do ... end
```

separate entities other than formal arguments may not appear as arguments in assertions. This is enforced by the following rule.

Definition 4.2.11 (Assertion Argument rule) *If an assertion contains a function call, any actual argument of that call must, if separate, be a formal argument of the enclosing routine.*

One of the consequences of rule 4.2.11 is that assertions appearing in class invariants may not use separate arguments. (There is no enclosing routine for a class invariant.)

Importing object structures

Function *clone* from *ANY* cannot be used to obtain copies of separate objects; since it is declared as

clone (*other*: *ANY*): **like** *other*

an attempt at using it with a separate actual argument would violates the rule SC1. SCOOP_97 provides another function for cloning separate object structures without producing traitors:

deep_import (*other*: **separate** *ANY*): *ANY*

The result is a non-separate object structure, recursively duplicated from the separate structure starting at *other*. In a way, *deep_import* corresponds to the *deep_clone* operation (see [94], p. 247) but the result is placed on the client's processor. Since all the object copies are non-separate, there are no traitors.

4.2.6 Proof rule for feature calls

The sequential character of processors and the FIFO scheduling of separate calls targeting the same supplier object — or indeed different objects but handled by the same processor — permits the derivation of a proof rule based on the sequential proof technique.

Definition 4.2.12 (Sequential proof technique) *Consider the call x.f (a). If we can prove that the body of f, started in a state satisfying the precondition, terminates in a state satisfying the postcondition, then we can deduce the same property for the above call, with actual arguments* a *substituted for the corresponding formal arguments, and every non-qualified call in the assertions (of the form* some_property*) replaced by the corresponding property on x (of the form* x.some_property*).*

The proof rule for feature calls (4.2.1) is an adaptation of Hoare's proof rule for procedures; however, only the non-separate preconditions and postconditions are used in its conclusion.

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{Pre_r^{nonsep}[\overline{a}/\overline{f}]\}\ \ x.r(\overline{a})\ \ \{Post_r^{nonsep}[\overline{a}/\overline{f}]\}} \tag{4.2.1}$$

To prove a routine correct, the same conditions must be demonstrated as in the sequential case, but the consequences on the properties of the call are different; the client has fewer properties to ensure before the call (just the non-separate preconditions) but it obtains fewer guarantees in return (just the non-separate postconditions). The former difference may be viewed as good news for the client, the latter as bad news.

4.2.7 Advanced features

To complete the description of SCOOP_97, we present two advanced mechanisms introduced in OOSC2 but not considered in our concurrency framework.

Duel mechanism

To provide support for ABCl/1-style *express messages* [148], SCOOP_97 lets an impatient client snatch a shared object — more precisely, the processor that handles the object — from its current holder. If a client is holding a lock on a supplier's handler for too long, at the detriment of another client judged more important, the latter may interrupt the current holder through a call to the library routine *demand*. A successful call will cause an exception in the current holder, which must have accepted the possibility in advance; the holder releases the lock and handles the exception (usually by trying again later on). If the interrupt fails — because the holder has not agreed to yield — the exception will happen in the challenger object; the holder retains its lock. This *duel* mechanism provides added flexibility without violating the other rules of SCOOP_97 or the requirements of Design by Contract. It enables the implementation of real-time facilities, e.g. a watchdog (see examples in [94], p. 1019).

The duel mechanism is beyond the scope of this dissertation; we do not discuss it in the critique part below. Real-time programming with SCOOP is a topic of another PhD project in our group [9]; that project also refines the duel mechanism to make it more flexible, e.g. by introducing multiple priority levels and timeouts on duels.

Mapping of processors to physical resources

Since processors are abstract and the program text does not specify what physical resources they should be mapped to, this specification must appear elsewhere. SCOOP_97 uses a *Concurrency Control File* (CCF) which describes the available computing resources — CPUs, servers, etc. — and specifies the mapping between processors and physical resources. CCF files are separate (in the usual sense of the word) from the software. An application may be compiled without any reference to a specific hardware or network architecture; at run time, each component of the application will use a CCF to find out about the available local and remote computing resources.

The customised mapping of processors to physical resources is beyond the scope of this dissertation; so is the use of distributed architectures. (Distributed programming with SCOOP is a topic of other projects in our group; see the discussion of future work in chapter 13.) Our implementation (see chapter 11) uses a single machine where a single thread represents each processor; we assume that there is no limit on the available number of threads.

4.3 Related work

This section presents related work by other authors and reports on the previous implementation attempts.

Compton's runtime

Compton [43] describes a runtime for SCOOP_97, based on the SmartEiffel [2] compiler (called SmallEiffel at that time). A semi-formal semantic description, based on a similar model for Eiffel// [14], is also proposed. The author performs an interesting analysis of the model and points out several inconsistencies. Nevertheless, to preserve the basic model, only minor corrections are carried out. The rules for expanded types are more stringent: if the result of a separate call is expanded, then its base class must not declare any once functions or non-separate attributes. (SCOOP_97 only requires that the base class must not contain any non-separate attributes.) To eliminate an inconsistency in the treatment of separate calls, the author suggests to modify the separate call rule 4.2.2 so that the target of a separate call, *except a creation call*, must be a formal argument of the routine in which the call appears. The treatment of once features is also restricted to ensure the absence of traitors: all once features have the *once per processor* semantics, i.e. their result is shared by all instances of the same class handled by the same processor. This causes several problems, e.g. it is impossible to properly type the entity *io* which denotes the standard input-output console; *io* is non-separate although it is shared by all objects in a system. Our approach supports an unconstrained use of once functions (see section 9.4): the once per processor semantics is applied to functions that return a non-separate result; the *once per system* semantics applies to functions that return a separate result, i.e. their result is shared by all instances of a given class within the system. Also, we do not impose any restrictions on expanded types (see section 6.10).

Compton describes SCOOP_97 synchronisation as an instance of the mutual exclusion problem [78, 22]. A processor's execution cycles through four stages: *entry*, *critical*, *exit*, and *remainder*. The *entry* stage corresponds to lock acquisition and wait condition checking. The *critical* stage is the execution of a routine's body. The *exit* is the lock release operation. (Postconditions are ignored; in our framework, the *exit* stage would also include postcondition checking.) The *remainder* are all non-separate calls before and after the call to an enclosing routine.

The locking algorithm lets a processor overtake another one that is suspended on the wait queue of a resource. The author justifies this choice with a reduced potential for deadlock. We agree that the approach may reduce the number of deadlocks but it also violates the fairness guarantees of SCOOP_97, in particular the FIFO policy for servicing clients' requests. As a result, starvation may occur. A common point of Compton's lock manager and our scheduler (see section 11.2) is that they do not permit concurrent locking of non-intersecting sets of resources requested by different processors. While *recursive locking* is used in his model, i.e. a processor that holds a lock on another processor may request and get another (redundant) lock, we use no recursive locking in SCOOP; if the requested processor has already been locked by the requesting processor, no additional lock requests are issued.

An interesting point is the detection of a system's exit condition; it has not been considered in SCOOP_97 at all. The system should exit when all processors enter a dormant state. Since there is no way of detecting whether all processors are dormant — because there is no "supervisor" thread in Compton's runtime system — each processor that wants to enter a dormant state has to check whether all the others are dormant; if yes, the whole system exits. We propose a different solution, based on the supervision by the scheduler (see section 11.2). Whenever the scheduler suspects quiescence, i.e. a state where all processors are dormant, it checks whether all processor are indeed idle; if yes, it asks them to terminate and it terminates itself, which

[2] http://www.smarteiffel.org

results in the system's exit. Checking the state of all processors is a costly operation, so the scheduler only performs it when there are no pending requests and no requests are currently being serviced. As a result, practically no run time overhead is incurred.

Compton's work is a first attempt at implementing SCOOP_97. Because of several restrictions, the lack of support for several advanced features, and the incompatibility of the applied locking mechanism with SCOOP_97, it may be viewed as a partial implementation only. Nevertheless, the semantics description has clarified several fuzzy points of the original model.

SECG

The SCOOP-to-Eiffel-Generator (SECG) Fuks et al. [60] translates SCOOP_97 code into pure Eiffel code with embedded calls to the EiffelThread library. SECG does not require any changes to the EiffelStudio compiler or runtime, and all Eiffel programming constructs can be used; this is a big advantage over Compton's system. Mutexes and buffers are added to separate classes in order to keep track of pending call requests made by clients. Similar changes are made to separate entities and separate arguments to introduce mutexes, allowing synchronisation and mutually exclusive access. Each separate class, when translated, inherits from the library class *THREAD* and is provided with a buffer containing pending feature call requests. The root class of the system executes all threads; each thread, indefinitely, retrieves a pending request from the queue and executes the corresponding feature. SECG translates SCOOP_97 programs following the schema below.

- Class *EXCEPTIONS* becomes the ancestor of all the user-defined classes in the system.
- Class *THREAD_CONTROL* becomes the ancestor of the root class. *THREAD_CONTROL* provides the basic features for controlling the execution of threads. The root class is enriched with the feature *request_pending* which acts as a resource monitor for the root object, and *request_pending_mutex* which synchronises the accesses to the monitor. The root class is further extended with the features *is_request_pending* and *rescue_SCOOP*. The latter provides a handler for concurrency-related exceptions.
- Each separate class becomes a thread; it inherits from the library class *THREAD*. A separate class is then enriched with additional features implementing a *request buffer* and its synchronisation policy.
- Every declaration of the form x: X, where X is a separate class, is replaced with

  x: X
  x_mutex: *MUTEX*

  MUTEX is a class from EiffelThread. A similar substitution is performed on the declarations of formal arguments.
- Since new attributes are introduced in several classes — separate ones and those that use separate attributes — creation procedures have to be extended accordingly. In particular, *request buffers* and *mutexes* have to be created.
- Separate calls and calls that take separate arguments are translated into calls which register corresponding feature requests, e.g. $x.f\ (a)$, where x and a are separate, becomes

$$x.\ set\_feature\_to\_do\quad([p,\ ''F\_STRING'', a,\ a.mutex])$$

The first element of the tuple represents the target of the call; the second is a string representing the requested feature; the last two elements represent the actual argument of the feature call, and the mutex used for synchronising accesses to it.

The translation is straightforward. Nevertheless, since every object represented by a separate entity has its own handling thread, mutex, and a request buffer, SECG is only applicable to programs where each processor has a "main" object which communicates with the main objects of other processors, very much in the style of CEE [70]. It is not possible to reference different separate objects located on the same processor. Furthermore, although separate entities may become attached to non-separate objects, all calls on such entities are treated as strictly separate. For example, the following source code

```
x: separate X
non_separate_x: X
...
create non_separate_x
x := non_separate_x
x.f
non_separate_x.g
```

is translated in such a way that $x.f$ is treated as a separate call, hence executed asynchronously. (It results in x's request queue being extended with a request to execute f.) The subsequent call to $non\_separate\_x.g$, targeting the same object, is synchronous and will be executed before $x.f$; this may violate the assumed execution order.

Separate local variables are not supported. The justification is that "an entity declared as **separate** is intended to be shared by multiple threads; thus it seems that declarations of **local** and **separate** are incompatible." The SCOOP framework presented in this dissertation permits the use of separate local variables. They may be used for the creation of new separate objects or for the temporary storage of separate references; the lifetime of a local entity corresponds to that of its routine but the object represented by the entity often outlives the routine.

There are no further limitations with SECG: any valid Eiffel constructs, including once routines and expanded types can be used. Postconditions are checked before releasing the locks acquired by the routine. Therefore, SECG allows separate calls in postconditions, and it applies the sequential (blocking) semantics to such assertions. This simplifies the treatment of contracts and exceptions but it also increases the likelihood of deadlock and limits the amount of parallelism (see section 5.6). Surprisingly, separate calls are also allowed in class invariants; however, the rationale for such invariants remains unclear.

The work on SECG has raised many interesting questions which have fuelled our research; in particular, it has prompted us to clarify the semantics of separate annotations and the treatment of assertions.

SCOOP for SmartEiffel

Adrian [2] describes an attempt at implementing SCOOP_97 in SmartEiffel. Despite its incompleteness — large parts of the draft are missing and it is not clear how much of the planned

implementation has been done — the document provides a few interesting insights. Starting from Compton's description [43], the author proposes solutions to several problems, e.g. once queries, conformance of separate and non-separate entities, and exceptions.

Adrian clarifies the relation between separate and non-separate entities in the following manner: *A* conforms to **separate** *A*; furthermore, if *B* conforms to *A*, then

- **separate** *B* conforms to **separate** *A*.
- *B* conforms to **separate** *A* (by transitivity).

This corresponds to the non-strict semantics of **separate** annotations (see section 5.1). Our type system captures the conformance relation more precisely (see section 6.3).

The treatment of once functions is different from Compton's: a once per system semantics is preferred here, as it purportedly leads to a separate evaluation, whether the function is really separate or not. The author claims it to be the only sound and practical solution. (Which brings the number of "the only sound solutions" to two, if you believe both Compton and Adrian.) In section 9.4, we show an alternative solution which combines both approaches.

To handle exceptions in a concurrent context, Adrian suggests that an exception reaching the boundary of a processor should mark that processor as "dirty", i.e. unable to serve any request. If a client is waiting for the dirty processor because of wait by necessity, an exception is propagated to the client; furthermore, any subsequent call on an object handled by a dirty processor immediately raises an exception in the client. A recent proposal by Arslan [11] and Meyer follows a similar approach.

Bailly's semantics and proof system

Bailly [16] proposes an operational semantics for a subset of SCOOP_97, and a set of rules for the inference of safety properties of concurrent programs. The author assumes a different semantics of separate preconditions — they are merely guards of conditional critical regions (CCRs) represented by routine bodies. Guards are excluded from contracts and treated separately from traditional (correctness) preconditions. The approach does not support inheritance, therefore problems caused by guard strengthening vs. precondition weakening are not discussed. The treatment of postconditions is identical in sequential and concurrent contexts. Following the CCR semantics, the unlocking of separate objects locked by a routine is performed atomically. As a result, it is impossible to reason about features that involve separate callbacks. Additionally, query calls may only appear at the end of a routine's body.

Bailly discusses the infeasibility of formal reasoning with the proof rule 4.2.1; a non-compositional proof method is sketched along the lines of the proof system for concurrent Java programs [1]. As opposed to Java, no additional class invariants (in Owicki-Gries style [115]) are necessary to ensure the interference-freedom because intra-object concurrency is prohibited. Nevertheless, the presence of asynchronous calls increases the complexity of proofs; each asynchronous call requires three cooperation tests, whereas a synchronous call only requires two. The author views the CCR-like synchronisation as an extension of Java's **synchronized** blocks: in Java, only one object may be locked using a single **synchronized** block; SCOOP_97 offers the possibility to lock several objects at once. Modular reasoning about the proposed CCR primitive is difficult because locking and unlocking typically involves the information

about the environment of the concerned objects. This makes the use of a global CCR invariant unavoidable; no such invariant is necessary for Java programs.

The treatment of routine calls as implicit CCRs falsifies the intended semantics of contracts and increases the complexity of the proposed proof system. Our proposal (see chapter 8) makes a full use of preconditions and postconditions instead of excluding them from the framework; this results in simpler proofs.

CSP semantics

Brooke et al. [34] propose a CSP semantics for SCOOP_97. The main goal is to obtain a better understanding of the model and to draw out the complexities and the unclear elements. The CSP model (Communicating Sequential Processes) [129] provides much of the infrastructure needed to capture concurrency and synchronisation mechanisms; also, powerful tools for analysing CSP specifications, i.e. FDR [87], already exist. The following elements of SCOOP_97 computation are targeted:

- object reservations: what happens when client c reserves y, then c calls x which itself reserves y?
- preconditions,
- lazy evaluation, i.e. wait by necessity,
- exception propagation: what happens if the caller has already finished?
- underlying communications and scheduler structure.

In addition to objects and processors (referred to as *subsystems*), the formal model includes *partitions* which represent physical computational resources: CPUs, POSIX processes, or individual threads. Each partition is responsible for zero or more processors. This approach follows the Ada 95 model of distributed processing [137]. There is no object migration between processors and no processor migration between partitions. An implementation of partitions must ensure that no processor is starved; essentially, each partition must provide cooperative multitasking.

Separate preconditions are translated directly into CSP guards; since the study does not consider inheritance and the related problem of precondition weakening vs. guard strengthening, this mapping does not cause any particular problems.

When locking separate arguments $\overline{a}$, an atomic test-and-set routine is necessary: if all the elements of $\overline{a}$ are available, they are reserved atomically. Because the different elements of $\overline{a}$ might be on different partitions, the authors suggest a "big lock" approach, where each reservation request acquires a lock on the entire system before checking the availability of the requested resources. Although not good for performance, this is the semantics that the overall system should exhibit; it is suggested that practical implementations may use a different approach provided that it preserves this semantics. (SECG [60] indeed uses the big lock approach; YO_SCOOPLI [111] does not have an explicit global lock but the global queue of requests provides a similar guarantee; Compton's scheduler [43] uses a weaker semantics that does not ensure fairness.)

Brooke et al. provide a solution for asynchronous exceptions: whenever an exception needs to be propagated to a separate client, the next caller that queues a separate call on the "dirty" object (the object that executed the faulty routine) should receive an exception of a special type *separate_routine_failure* . The implication of this proposal is that queueing a call on a request queue of a processor is synchronous.

The need for lock passing is identified. Locking is transitive by default, i.e. if a client object c holds locks on supplier objects x and y, and x requests a lock on y, x acquires that lock, independently of whether c allows it or not. (Our approach, described in section 7.2, lets the client decide whether a lock should be passed.) Furthermore, no synchronisation between the calls on y issued by c and those issued by x takes place, i.e. it is possible that they are interleaved. This offers more potential parallelism than our solution but reasoning about programs becomes intractable because one has to account for all potential interleavings. Additionally, it is possible for client c to unlock y before x revokes its lock on y. Our mechanism prohibits such behaviour in order to guarantee atomicity; locks that have been passed to a supplier must be revoked before the client proceeds. The formal model proposed by Brooke et al. may be extended to account for the differences mentioned above (see section 7.3).

The CSP formalisation and the ensuing discussion — in particular on the mapping of objects to processors, and the lock passing mechanism — have been invaluable for the development of our framework.

Atomic features

Vaucouleur and Eugster [142] propose an alternative approach to synchronisation, based on the concept of transactions [61]. Features may be declared as *atomic*; such features have the "all-or-nothing" semantics, i.e. they either execute to completion, or do not execute at all. If the body of an atomic feature cannot be executed completely, the transaction is rolled back or compensated. Since atomic features operate on a copy of the object structure — different atomic features use different copies — no locking is necessary; the rule 4.2.4 of SCOOP_97 does not apply.

This approach seems to be particularly suited for short transactions with little potential for conflicting accesses; modelling longer operations, in particular those involving several cooperating processors, cannot be efficiently represented in that way. It is not clear whether the transactional approach may be combined with locking to cater for the needs of both conflicting and cooperative concurrency; the most problematic point is the treatment of separate assertions.

A recent feasibility study [101] demonstrates that atomic features can be used in small systems running on a single machine but the approach has not been tested on large applications. Also, no distributed implementation exists yet.

Inheritance-based implementation

Since no satisfactory implementation of SCOOP_97 was available at the outset of this work, we started by implementing the model. Our initial implementation [111] — YO_SCOOPLI (where YO stands for "Ye Olde") — used multiple inheritance to implement the separate semantics of objects and classes. The criteria for the library design were to make it as simple and easy to use as possible, and to maintain a clear correspondence with the original SCOOP_97 syntax.

YO_SCOOPLI implements the basic SCOOP_97 mechanisms: separate calls, wait by neces-

sity, locking through argument passing, and wait conditions. Nevertheless, it is only possible to pass separate or expanded objects across processors' boundaries; non-separate references cannot be used as arguments or results of separate calls. Also, no attachments from non-separate to separate entities are possible. These two problems are due to the reliance on inheritance to implement separateness; conceptually, **separate** *S* is a supertype of *S* but the inheritance relation between the class *SEPARATE_S* (that simulates **separate** *S*) and *S* goes in the opposite direction. (This also enforces the strict semantics of **separate**.) The library does not support separate calls on queries implemented as attributes. The necessity to declare a dedicated feature for each expanded type is a serious limitation of the library; such features are provided for the most commonly used types — *BOOLEAN*, *INTEGER*, *CHARACTER*, *REAL*, and *DOUBLE* — but there is no support for user-defined expanded classes.

Although YO_SCOOPLI only provides a limited support for SCOOP_97 constructs, it is worth mentioning as our first practical attempt at understanding the internal workings of the model. Many mechanisms present in the current implementation, such as the communication between processors, the scheduler providing the strong fairness guarantees, and the use of agents to represent feature requests, have been adopted from this library (see section 11.2 for details). YO_SCOOPLI has also demonstrated that the conformance relation between separate and non-separate types cannot be simulated and implemented using inheritance; this impossibility result has prompted us to look closer at the problem of separate semantics and suggested a relation between separate annotations and the type system.

5

Beyond SCOOP_97: critique and roadmap

DUE to its simplicity and the integration of concurrency with O-O concepts, SCOOP_97 is an important first step towards a practical method of concurrent programming. Nevertheless, the impact of concurrency on the semantics of O-O mechanisms, in particular contracts, has not been studied in enough detail; it is a source of inconsistencies and limitations. Advanced mechanisms — polymorphism, dynamic binding, genericity, and agents — have not been tackled at all. Several validity rules are too restrictive; they need rethinking. So does the semantics of contracts; it has to be refined to make DbC usable in a concurrent context, both as a basis for proofs and as a design tool.

This section discusses the problems found in SCOOP_97 and points out the limitations of the model. It constitutes a roadmap for the development of the current SCOOP model described in chapters 6 – 10; each identified problem is solved there to provide a consistent, expressive, and usable concurrency framework.

5.1 Semantics of separate annotations

Entities decorated with the keyword **separate** denote objects handled by a different processor than the current object. But it is not possible to ensure this property statically; since SCOOP_97 does not prohibit direct attachments from non-separate to separate entities, the assignment

```
my_x: separate X
x: X
 ...
my_x := x
```

immediately results in *my_x* being attached to a non-separate object. Even if we introduced a rule to prohibit such assignments, it would be possible to violate the semantics of **separate** annotations. Consider the example in figure 5.1. After the creation instruction **create** *my_x*, the new object attached to *my_x* is indeed separate, i.e. it is placed on a fresh processor (different from the current one). The assignment *my_y* := *my_x* preserves the semantics of **separate**; the object represented by *my_y* is now handled by a different processor than **Current**. On the other hand, the call *r* (*my_x*) violates the intended semantics: through the call *x*. *set_y* (*my_y*), the expression *my_x.my_y* now denotes a non-separate object (*my_x* itself) although *my_y* is declared as **separate** in class *X*. (Note that no attachment from non-separate to separate has been used here.)

```
-- in class C
my_x, my_y: separate X
r (x: separate X)
    do
        x.set_y (my_y)
    end
...
create my_x
my_y := my_x
r (my_x)

-- in class X
my_y: separate X
set_y (y: separate X)
    do
        my_y := y
    end
```

Figure 5.1: Problems with the semantics of **separate**

There is a clear mismatch between the strict semantics of **separate** requiring that the objects referenced by separate entities *must* be handled by a different processor, and the intuitive non-strict semantics that allows such possibility but does not enforce it, i.e. separate entities *may* represent separate objects or, expressed differently, separate entities denote *potentially separate* objects. The separate class annotations, the separate call rule (4.2.2), and the wait semantics of separate preconditions clauses follow the strict semantics, whereas the consistency rules SC1–SC4 (4.2.6 –4.2.9) follow the non-strict one; this leads to numerous complications and inconsistencies.

It is necessary to apply the same semantics in all rules and mechanisms of the model; we opt for the non-strict semantics because it is the only practical solution: it is enforceable statically and compatible with the modularity requirements. Also, it removes the inconsistencies and simplifies the type system (see chapter 6). Most related work on SCOOP_97 assumes the strict semantics [60, 34, 16]; Compton [43] and Adrian [2] are notable exceptions.

Separate Current paradox

SCOOP_97 permits the use of the **separate** keyword to decorate class headers. Every instance of a separate class *C* is separate from all other objects in the system. But it is impossible to ensure such a property statically; this is a sufficient reason to abolish the separate class annotations. Nevertheless, there exists an even more fundamental reason for eliminating these annotations: they lead to a logical inconsistency we call the *separate Current paradox*. In a separate class, is **Current** separate or non-separate? According to the separate class semantics, **Current** is separate; but **Current** denotes the current object, so it cannot be separate! Figure 5.2 illustrates the problem. If **Current** is separate then, following the separate call rule (4.2.2), the call to **Current**.*f* is invalid because its target is not a formal argument on the enclosing routine. Similarly, the call to *f*, although appearing in an unqualified form, implicitly targets **Current**;

```
separate class C
feature
    r (c: C)
        do
            c.f
        end

    f do ... end

    ...
        Current.f    -- Invalid?
        f            -- Invalid?
    ...
end
```

Figure 5.2: Separate **Current** paradox

therefore, it is invalid. One could think of a special rule for **Current**, or at least for unqualified calls, but this would further complicate the model. Since the separate class annotations do not bring any real benefit in terms of expressiveness and convenience, the introduction of such a rule is not justified.

The use of separate classes also complicates the support for inheritance and polymorphism. If a separate class C inherits from a non-separate class B, should it conform to B, i.e. should clients be able to perform polymorphic attachments from C to B? Such attachments would introduce potential traitors, so they must be prohibited. This would add yet another special rule to the model, unnecessarily complicating the conformance relation among classes. Therefore, the use of separate annotations on classes should be prohibited.

5.2 Separate call rule

Rule 4.2.2 enforces a strict access control policy: only formal arguments may be used as targets of separate calls. The rule is simple — both for the programmer and the compiler — because it uses a *syntactic* distinction between valid and invalid targets. Nevertheless, the policy is too stringent: several interesting synchronisation scenarios are rejected even though they do not violate the mutual exclusion requirement. Consider the example in figure 5.3. The local variable y may not serve as target of separate calls because it is not a formal argument; but y denotes the same object as x, and calls on x are permitted! Even more disturbing is the impossibility of using a separate expression as target of a call, e.g. the call $x.g.f$ is prohibited because $x.g$ is not a formal argument of r; this call is obviously safe because the object denoted by $x.g$ is handled by the same processor as x (g returns a non-separate result). Using the local variable z to store the value of $x.f$ — thus getting rid of the multidot form — does not help: the call $z.f$ is invalid for the same reasons as $y.f$.

The syntactic distinction between formal arguments and other entities fails to capture precisely the intended *semantic* requirement: what only want to allow calls on objects handled by processors currently held (locked) by the client. Formal arguments are just a subset of all safe

```
r (x: separate X; a: A)
    local
        y, z: separate X
    do
        x.f        -- Valid
        y := x
        y.f        -- Invalid although safe
        x.g.f      -- Invalid although safe
        z := x.g
        z.f        -- Invalid although safe
        a.f        -- Valid
        s          -- Valid
    end

s do ... end

-- in class X
g: X do ... end
```

Figure 5.3: Limitations of the separate call rule

targets; other safe targets, such as multidot expressions and local variables that reference objects handled by locked processors, should also be permitted. The expressive power of separate annotations is not sufficient to capture all properties required by such a rule. We need some way to assert that two entities represent objects handled by the same processor, so that calls $y.f$ and $z.f$ in figure 5.3 can be recognised as safe by the compiler. Section 5.5 discusses this topic in detail, and proposes a type-based solution.

If we push this line of thinking to the extreme and focus solely on the semantic requirement rather than syntactic distinctions of any kind, it turns out that a *separate* call rule may not be necessary at all; it could be integrated into a general call rule applicable to separate and non-separate calls. This is because there is no good reason to differentiate between separate and non-separate entities: the real difference is between entities handled by processors under our control and those beyond our control. Non-separate entities denote objects handled by the current processor; this processor is under the client's control in any context. From the correctness point of view, there is no difference between the call to $x.g$ and the calls $a.f$ and s in figure 5.3; all of them target entities handled by a processor locked in the current context. A generalised call rule could be based on this locking requirement; it needs to be combined with the requirement of non-voidness of targets (rule 8.23.14 /VUTA/ in [53]).

5.3 Feature call vs. feature application

The wait rule (definition 4.2.4) enforces the synchronisation through argument passing and separate preconditions *at the time of the call*, and requires arguments to be reserved by the client's handler. It is easy to see that this rule is flawed: in the following code excerpt

```
my_a: separate A

r (x: separate X)
    do
        x.f (my_a)
        next_instruction
    end

-- in class X
f (a: separate A)
    require
        a.some_property
    do
        a.r
        ...
    end
```

the call $x.f\ (my\_a)$ forces the client to wait because *my_a* has to be reserved on its behalf and the separate precondition of f must hold at the moment of the call. The client's handler cannot move to *next_instruction* before f has terminated; this results in an unintended synchronous semantics of the call. Additionally, *my_a* is reserved for the client, although the calls on the corresponding formal argument a in the body of f are performed by the supplier x. This is clearly not what we want; the client should proceed immediately with the execution of *next_instruction* because f is a command; the reservation and the establishment of the precondition should happen on the supplier's side *at the moment of the feature application*; *my_a*'s handler should be locked on behalf of x.

The confusion is probably due to the fact that all the examples discussed in [94] and other articles involved only two forms of feature calls: a non-separate call of the form $r\ (a)$ where a is separate, and a simple separate call $x.f$ where f either takes no argument or its arguments are expanded (thus require no locking). No attempt was made at defining the semantics of $x.f\ (a)$ where both x and a are separate. This led to the incorrect amalgamation of the feature call and the feature application mechanisms.

These two fundamental concepts should be clearly distinguished, even in the case of non-separate calls; the only particularity of a non-separate call is that the feature call and the feature application are performed by the same processor. Hence the synchronous semantics of such calls: the feature request is immediately followed by the feature application; the client cannot proceed before the feature has terminated even if the feature is a command. Nevertheless, non-separate calls should be seen as a particular case of the general asynchronous call mechanism. This clarifies and simplifies several language rules; it also constitutes a convenient basis for the unification of the contract semantics discussed in chapter 8. See section 6.1 for a precise definition of the feature call and the feature application semantics and a detailed discussion of the related topics.

5.4 Consistency rules

The four separate consistency rules SC1–SC4 (4.2.6 –4.2.9) should eliminate potential traitors. The rules are easy to understand and apply. But are they sound, i.e. do they really prohibit the introduction of traitors? On the other hand, do they not introduce unnecessary constraints limiting the expressive power of the language? Let's have a closer look at each rule.

SC1 eliminates the most obvious source of traitors: a direct attachment from a separate to a non-separate entity. At the same time, it does not prohibit attachments in the opposite direction. Therefore, it captures the intuitive conformance between *T* and its separate counterpart **separate** *T*. It seems to be sound; it does not impose unnecessary constraints.

SC2 requires the formal argument of a routine to be declared as separate if the corresponding actual is separate. The rule is sound but overly restrictive: even if an actual argument is shown to be handled by the same processor as the target of the call, the call is invalid unless the formal argument is separate. Figure 5.4 illustrates this situation: the call *l*.*extend* (*s*) is invalid although *s* is non-separate from *l*. This is a very common scenario, in particular when reusing a "sequential" library class, such as *LIST*, in a concurrent context (see section 5.12.3 for the discussion of code reuse). Even worse: the call *l*.*merge* (*l*) is invalid too, although *l* is certainly non-separate from itself! One could argue that it is sufficient to mark formal arguments of all routines as separate to satisfy the rule. Unfortunately, most routines are meant to take non-separate arguments; they would be meaningless with separate formals. Therefore, rule SC2 needs to be relaxed.

```
r (l: separate LIST [STRING])
    require
        not l.is_empty
    local
        s: separate STRING
    do
        s := l.i_th (1)
        l.extend (s)              -- Invalid
        l.merge (l)               -- Invalid
        l.append (l.i_th (1))     -- Invalid
    end
```

Figure 5.4: Limitations of rules SC2 and SC3

SC3 mirrors SC2: it requires that the target of an attachment be separate if its source is a result of a separate call to a function returning a reference type. The unnecessary restrictions imposed by SC3 are similar to those of SC2; consider the call *l*.*append* (*l*.*i_th* (1)) that should be allowed because *l*.*i_th* (1) is certainly non-separate from *l*. A minor mistake in SC3 — the fact that it only mentions functions but not attributes — is a source of unsoundness.

Rule SC4 takes care of separate objects being passed across processors' boundaries: such objects must be instances of fully-expanded classes, i.e. classes without non-separate attributes of a reference type. It seems that non-separate functions should also be considered here but it is not necessary: the only way for them to produce a result is to create a fresh object or to get it by applying a query to a non-separate attribute; in both cases, no traitor is created.

So, the rule is sound. Nevertheless, the full-expandedness requirement is too restrictive because fully-expanded types are rare. In practice, only the basic classes *BOOLEAN*, *INTEGER*, *CHARACTER*, *REAL*, and *DOUBLE* may be used; user-defined expanded classes are unusable because they usually contain non-separate attributes. Rule SC4 must be relaxed to support unconstrained use of all expanded types.

The consistency rules have been designed in an ad-hoc manner: a new rule is added whenever a source of traitors is identified that has not been eliminated by previous rules. That is why rule SC1 is the most general and it captures most traitors, whereas rule SC4 covers the most specific (and rare) scenario involving expanded objects. The obvious danger of this approach is that some sources of traitors may be omitted, simply because the rule designer has not considered a particular scenario. A closer look at the consistency rules reveals that, even though SCOOP_97 does not view separateness as a type property, the rules try to capture a conformance relation between non-separate and separate types. Therefore, we should aim at integrating the separateness property into the type system, so that informal rules written in a natural language be replaced by precise formal type rules whose soundness can be demonstrated, and which can be readily used by a compiler. An enriched type system for SCOOP is presented in chapter 6.

5.5 Reasoning about object locality

Programmers should be able to assert that the object represented by a given entity are separate or non-separate from the object represented by another entity. But **separate** annotations are not sufficient: one can only assert the locality of an entity with respect to **Current**; the relative locality of two entities cannot be asserted and reasoned about. The critique of separateness consistency rules SC2 and SC3 in section 5.4 has pointed out the weaknesses of the type system that limit the expressiveness of the model; the problem is caused precisely by the insufficient expressive power of **separate** annotations. Let's have look at figure 5.4 again: it is impossible to assert that s is handled by the same processor as l, hence the call $l.extend\ (s)$ is invalid. Similarly, the reasoning about the locality of $l.i\_th\ (1)$ is imprecise: the compiler only knows that it is separate from **Current** but the knowledge about its non-separateness from l is lost; as a result, the call $l.extend\ (l.i\_th\ (1))$ is invalid.

We should be able to assert the property "s is separate from **Current** but non-separate from l". Also, the compiler should make use of the obvious fact that each non-writable entity, such as the formal argument l, is non-separate from itself; so are results of non-separate queries applied to such entities, e.g. $l.i\_th\ (1)$. The enriched type annotations proposed in this dissertation express the relative locality of objects in a compact way (see section 6.2.2).

Separate to non-separate downcasts

When a separate entity becomes attached to a non-separate object, the information about the object locality is lost; even though we know that an entity denotes a non-separate object at run time, we cannot assume it at compile time. SCOOP_97 provides no mechanism to perform an appropriate run-time check. Eiffel's *object test* mechanism may be extended to account for separateness; this would provide a convenient way to downcast separate entities to non-separate ones (see section 6.7).

5.6 Semantics of contracts

Preconditions

SCOOP_97 uses two different semantics for preconditions, depending on whether they involve separate calls. Preconditions involving separate calls have wait-semantics, i.e. they cause the client to wait if they are not satisfied, whereas non-separate preconditions preserve their correctness semantics, i.e. an exception is raised in a precondition is violated. This may be confusing because the same syntactic construct (the **require** clause) is used for two different purposes [27].

Since we claim that sequentiality is a particular case of concurrency, each sequential O-O mechanism should be simply a refinement of a more general concurrent mechanism. This principle also applies to preconditions: the wait semantics should be the general one, with the correctness semantics being its refinement. It sounds surprising at first but we may simply assume that every violated precondition causes the client to wait. If the precondition is separate, then the client may eventually be unblocked; if the precondition is non-separate, then the client will be blocked forever. But such a "deadlock" can be detected by the runtime (which would simply raise an exception). Therefore, all preconditions are conceptually the same, i.e. they are wait conditions. See chapter 8.1.1 for a detailed discussion of this topic. Such a generalisation of the precondition semantics solves many problems, e.g. the conformance of non-separate actual arguments to separate formals, and the use of polymorphism and dynamic binding (see section 5.10). It also facilitates the reuse of code (see section 5.12.3).

Postconditions

The sequential semantics is particularly problematic in the case of separate postconditions: they cause waiting, thus minimising potential concurrency and increasing the danger of deadlock, as illustrated in figure 5.5. The client wants to spawn some job at *york* and then continue its

```
spawn_two_activities (l: separate LOCATION)
    do
        l.do_job
    ensure
        l.is_ready
    end

york: separate LOCATION
...
spawn_job (york)
do_local_stuff
```

Figure 5.5: Problems with postconditions

own work represented by *do_local_stuff* . The client wants the guarantee that the job will eventually terminate; the most natural place to express such a guarantee is the postcondition of *spawn_job*. Unfortunately, since the postcondition clause is a query, its evaluation obeys

the wait by necessity principle; as a result, the client has to wait for the postcondition rather than moving immediately to *local_stuff* . This is a serious limitation; in fact, the benefits of parallelism are completely lost here. Now, assume that feature *do_job* needs the access to the client's processor. It requests a lock and blocks until the lock is granted; at the same time, the client is blocked waiting for the postcondition of *spawn_job*. This results in a deadlock.

In addition to the separate precondition paradox, Meyer [94] also mentions an interesting *separate postcondition paradox*: on return from a separate call, the client cannot be sure that the postcondition clauses still hold, even though they are guaranteed to hold when the call terminates. This is because the processor handling the involved supplier may become free in the meantime and other clients may have jump in and modify the state of the supplier, thus invalidating the postcondition. (The same problem is identified by Rodriguez et al. [128] as *external interference*; see section 3.2.) Applied to our example, this means that the client cannot make any use of *york*. *is_ready* , except for knowing that this property became true just before unlocking *york*'s processor. But this guarantee is given whether the client waits or not, so why should it wait? It may as well proceed with the execution of *local_stuff* as soon as the body of *spawn_job* has terminated. This suggests a different semantics of postconditions whereby the client does not have to wait but the supplier guarantees the postcondition when the features scheduled in the body of the routine have terminated. Naturally, this semantics must boil down to the standard correctness semantics when no concurrency is involved. We develop a postcondition semantics along these lines in section 8.1.2.

Checks and loop assertions

In SCOOP_97, loop assertions and **check** instructions obey the wait by necessity principle, i.e. their evaluation is blocking. The disadvantage of this solution is similar as in the case of postconditions: the potential for parallelism is limited, without any strengthening of provided guarantees. For example, a client executing the sequence of separate calls in figure 5.6 has to wait after $x.f$ for the evaluation of $x.some\_property$; this is probably not what the client really wants.

```
r (x: separate X)
    do
        ...
        x.f
        check x.some_property end
        x.g
        ...
    end
```

Figure 5.6: Synchronous check instruction

Section 8.1.4 proposes an asynchronous semantics of **check** instructions, which lets the client proceed immediately with the execution of $x.g$ while asserting that $x.some\_property$ must hold after $x.f$; the property is evaluated after $x.f$ but before $x.g$, as expected. Loop assertions can be treated similarly.

5.7 Proof rules

Separate assertions are excluded from the formal reasoning framework: the conclusion of the proof rule (4.2.1) omits them altogether. Therefore, a client has fewer properties to ensure before the call (just the non-separate preconditions) but it obtains fewer guarantees in return (just the non-separate postconditions). The former difference may be viewed as good news for the client, the latter as bad news. From the point of view of the programmer, the exclusion of separate assertions can only be viewed as bad news because it hinders formal reasoning about software, in particular if most (or all) assertions are separate, as it is often the case. Figure

```
-- in class BUFFER [G]
put (v: G)
    -- Store v.
    require
        not is_full
    ensure
        count = old count + 1
    end

-- in class C
store_two (buffer: separate BUFFER [INTEGER]; i, j: INTEGER)
        -- Store i and j in buffer.
    require
        buffer.count <= buffer.size - 2
    do
        --   {buffer.count <= buffer.size - 2}
        buffer.put (i)
        --   {True}
        buffer.put (j)
        --   {True}
    ensure
        buffer.count = old buffer.count + 2
    end

my_buffer: separate BUFFER [INTEGER]
...
-- {True}
store_two (my_buffer, 5, 10)
-- {True}
```

Figure 5.7: Infeasible proofs

5.7 illustrates the problem: since the precondition and the postcondition of *store_two* involve separate calls, they are excluded from the proof, i.e. the client has nothing to prove before the call *store_two* (*my_buffer*, 5, 10) but it also cannot assume anything after the call. One could argue that this occurs because *my_buffer* is not under the client's control (i.e. locked by the client) before and after the call; however, the same problem occurs in a context where the

concerned separate object is under the client's control, e.g. *buffer* in the body of *store_two*. The precondition of *store_two* may be assumed before the call *buffer*.*put* (*i*) — although it is not necessary because the proof rule does not require us to satisfy the precondition of *put* — but the postcondition of *put* is ignored (because it is separate, just like its target). Therefore, the client cannot assume anything after the first call to *put*. Similarly, the postcondition of *buffer*.*put* (*j*) is ignored. As a result, we are unable to prove the correctness of *store_two*: its postcondition is not implied by the property assumed after the second call to *put*. If the rule 4.2.1 considered these separate assertions, the properties inferred by both calls to *put* would be strong enough to imply the postcondition of *store_two*; the routine would be proved correct.

The proof rule for concurrent programs must be adapted to account for separate assertions; otherwise, Design by Contract will not be fully usable as a modelling and reasoning tool. Section 8.2 introduces a stronger proof rule which takes into account separate assertions and supports modular reasoning about concurrent software.

5.8 Locking policy

5.8.1 Eager locking

SCOOP_97 requires all separate arguments of a routine call to be locked before the call can proceed. This policy is unnecessarily restrictive and it increases the likelihood of deadlock. Consider the feature *r* in Figure 5.8. The handlers of *x*, *y*, and *z* must be locked by the client

```
r (x: separate X; y: separate Y; z: separate Z)
    require
        some_precondition
    local
        my_y: separate Y
        my_z: separate Z
    do
        x.f   -- separate call
        my_y := y
        x.g   -- separate call
        my_z := z
        s (z)
    end
```

Figure 5.8: Feature locking all its arguments

object before the body of *r* is executed. But is it really necessary to lock all of them? Let's see: the body of *r* contains two calls on *x*, therefore *x* needs to be locked. There is no way around it: to ensure atomicity we must guarantee that no other client is currently using *x*. On the other hand, *y* only appears on the right-hand side of an assignment; no calls on *y* are made. Similarly, *z* only appears as source of an assignment and as actual argument of a feature call. It seems that we only need to lock the processor that handles *x*; it is not necessary for *y* and *z* because the body of *r* does not involve any calls on them. So, there is too much locking than necessary.

Not only does the eager locking introduce more run-time overhead due to unnecessary waiting and resource acquisition, it might also be very dangerous as it often leads to deadlocks — the more resources a client requires, the more likely it is to get in a deadlock situation. There is also a practical point here: programmers do not have any control over what should and what should not be locked, so they cannot express all their design choices. The locking policy needs to be relaxed, so that only the necessary locks are acquired; we need to provide an adequate language mechanism to capture programmers' choices. Section 7.1 proposes a refinement of the locking policy along these lines.

5.8.2 Cross-client locking and separate callbacks

The client performing a call to a routine that locks separate objects holds exclusive locks on these objects for the duration of the call. As pointed out in section 4.2.3, this policy ensures that no other client can jump in and modify the state of a supplier object between two consecutive calls. While such a guarantee is convenient for reasoning about concurrent software, it unnecessarily limits the expressiveness of SCOOP_97 and leads to deadlocks. Figure 5.9 illustrates the problem. Calls $x.f$, $x.g$, and $y.f$ are asynchronous (f and g are commands); the client does

```
r (x: separate X; y: separate Y)
    do
        x.f
        x.g (y)     -- x waits for y to become available.
        y.f
        ...
        z := x.some_query    -- Current waits for x.
                             -- DEADLOCK!
    end
```

Figure 5.9: Deadlock caused by cross-client locking

not wait for their completion. Following the *wait by necessity* principle (see section 4.2.3), the client has to wait for the result of the query call $x.some\_query$. Because of the FIFO scheduling policy of separate calls on the same target, x's handler is not able to evaluate *some_query* before finishing all the previously requested calls on x. Unfortunately, this causes a deadlock because x's handler cannot execute $x.g\ (y)$ until it acquires a lock on y's handler; the latter is still locked by the client and it can only be unlocked once the execution of r's body is finished, but this may only happen after the call to *some_query* has terminated. So, we have a deadlock here: the client is waiting for x's processor and vice-versa; none of them will ever make any progress.

In fact, getting into a deadlock situation is even simpler. A client only needs to pass itself as actual argument to a separate query call, as in figure 5.10. Since feature g called on x needs to lock the processor that handles **Current**, it blocks until that processor is unlocked. But it will never be unlocked because it is waiting for the completion of the call to g. Again, we have a deadlock.

In the above examples, a deadlock occurs when the client waits for one of its suppliers. Since the client is waiting, it does not perform any operations on its suppliers; therefore, it makes no use of the locks it holds. If the client could temporarily pass these unused locks (the

```
-- in class C
s (x: separate X)
    do
        z := x.g (Current)  -- x waits for Current; Current waits for x.
                            -- DEADLOCK!
    end

-- in class X
g (c: separate C): INTEGER
    do
        ...
    end
```

Figure 5.10: Deadlock caused by a separate callback

lock on y in Figure 5.9, and the lock on **Current** in Figure 5.10) to its supplier x, the supplier would be able to execute the requested feature and return a result; we would avoid deadlock. We use that observation to develop a mechanism which solves the problems of cross-client locking and separate callbacks (see section 7.2).

5.8.3 Void separate arguments

Void arguments seem to be a minor problem at first but they wreak havoc in any attempt at implementing SCOOP_97. What happens if a separate formal argument is void? The wait rule 4.2.4 does not take this possibility into account; the model is underspecified here. Two straightforward solutions are applicable:

- Prohibit passing a void actual argument if the corresponding formal is separate. This needs to be monitored at run time; if an argument is void, an appropriate exception is raised (before any attempt to acquire locks). As a result, the routine does not execute at all, even if no actual calls would be performed on the void argument. YO_SCOOPLI [111] uses this approach.

- Ignore void arguments, i.e. only lock the processors that handle non-void separate arguments (if any). No special monitoring is required at run time: if a routine performs a call on the void argument, a standard "call on void target" exception is raised; if there are no calls on the argument, its voidness goes unnoticed. Compton [43] uses this technique.

The first solution is cleaner because it detects a potential problem closer to its source: a void separate reference is detected as soon as it is passed as argument. Nevertheless, this approach restricts the use of separate arguments as sources of attachments. The second approach gives more flexibility but problems are often detected far from their source: a reference may be passed as argument several times before a call is applied to it; this complicates debugging. No solution seems to cater for all the needs.

The recent introduction of *attached types* into Eiffel [53] suggests a better way to solve the problem. Attached types may be used to detect statically whether an argument is void or

not, and give an appropriate locking semantics in both cases. The advantages of both previous solutions — clarity and flexibility — may be combined (see chapter 7).

5.9 Quasi-asynchrony

A sequence of separate procedure calls, e.g. the three calls on *x* in figure 5.11, may be executed asynchronously; the client does not need to wait for their termination. Nevertheless, we cannot achieve full asynchrony here because at least one synchronisation event is required before that sequence of calls: it is the call *r* (*my_x*). This call blocks until the processor handling *my_x* has been locked for the exclusive use of the client's processor, as required by the wait rule 4.2.4. Therefore, one may only speak about *quasi-asynchrony*. This synchronisation overhead may

```
r (x: separate X)
    do
        x.f
        x.g
        x.h
    end

my_x: separate X
...
r (my_x)      -- Synchronisation point
...
```

Figure 5.11: Quasi-asynchrony in SCOOP_97

seem benign — the client needs to synchronise just once and then it is free to spawn as many asynchronous calls as it likes — but it is often necessary (or more convenient) to perform calls without *any waiting whatsoever*. Logging and mailing operations are good examples: if I want to notify my supervisor that I have finished a chapter of my dissertation, I just send him an e-mail (see figure 5.12). Similarly, logging the results of the latest football games should be done asynchronously. We are not interested in the exact timing of these operations; it does not matter whether they are performed immediately or delayed; the only important thing is that they should should not block the client and they should be done at some point. (We might want some additional guarantees, e.g. that two asynchronous calls on the same target be processed in the FIFO order.) Unfortunately, the calls to *send_message* and *log_event* are potentially blocking; the client waits for *mailer* and *log* to become available if they are currently used by other clients. This results in a quasi-asynchronous execution; there is no way to avoid it in SCOOP_97.

To increase the expressiveness of the model and make it practical, we need a mechanism for scheduling such calls with no waiting on the client's side. An obvious solution is to introduce a keyword decorating fully asynchronous calls, e.g. **asynch**, and associate a special semantics with it, as proposed by Brooke et al. [32]; but this complicates both the model and the language. Since we want to avoid complexity, we devise an agent-based mechanism which provides the required facility without changing the underlying model; see section 9.3.4.

```
log_event (l: separate LOG; s: STRING)
    do
        l.write (s)
    end

send_message (m: separate MAILER; address, message: STRING)
    do
        m.send (address, message)
    end

log: separate LOG
mailer: separate MAILER
...
send_message (mailer, ''bm@supervisor.org'', ''Finished chapter 5.'')
log_event (log, ''Switzerland vs. France 0:0'')
log_event (log, ''Germany vs. Poland 2:0'')
...
```

Figure 5.12: Quasi-asynchronous logging and mailing

5.10 Polymorphism and dynamic binding

A complete O-O model should support polymorphism, i.e. entities of type A should be able to represent objects of any type B that conforms to A. SCOOP_97 relies on the subtyping rules of Eiffel but it does not define precisely the conformance relation between separate and non-separate entities. Consistency rules SC1–SC4 seem to capture the following relation: if B is a subtype of A, i.e. B inherits from A, then

$$\begin{array}{ccc} A & \longrightarrow & separate A \\ \uparrow & & \uparrow \\ B & \longrightarrow & separate B \end{array} \tag{5.10.1}$$

Can standard Eiffel rules for feature redefinition be used, given this extended notion of conformance? Obviously, clients must not be cheated on in the presence of polymorphism and dynamic binding, i.e. the actual version of the feature chosen at run time must abide by the original contract known at compile-time. Let's see: Eiffel allows a covariant redefinition of result types of queries; therefore, taking into account the relation 5.10.1, an attribute x: **separate** X may be redefined into x: X. This is sound; no traitors may be introduced by such a redefinition. Nevertheless, "unseparating" an entity may cause some problems, as illustrated in figure 5.13. Routine r takes a separate formal argument x and uses it in a precondition; thus, the precondition has the wait semantics. The client using the routine passes a separate actual argument $a.x$; the contract is clear: if $a.x.some\_property$ is not satisfied, the client has to wait. After the polymorphic assignment $a := b$, however, the client is convinced that the actual argument of the subsequent call $r\ (a.x)$ is separate but it is not. The wait semantics will be applied nonetheless, which results in a deadlock if $a.x.some\_property$ does not hold. Here, the problem is caused by a wrong application of the wait semantics rather than an inappropriate conformance rule.

```
class A
feature
    x: separate X
     ...
end

class B
inherit A redefine x end
feature
    x: X
     ...
end

-- in class C
r (x: separate X)
    require
        x.some_property        -- Wait condition
    do
         ...
    end
a: A
b: B
 ...
r (a.x)        -- Valid
a := b         -- Polymorphic assignment
r (a.x)        -- Valid but leads to a deadlock
```

Figure 5.13: Problems with covariant attribute redefinition

Formal arguments may also be redefined covariantly in Eiffel; following the conformance relation 5.10.1, separate formals could be redefined into non-separate ones. Figure 5.14 illustrates it: routine *r* in class *A* takes a separate formal argument; its redefined version in class *B* takes a non-separate one. But this is problematic: clients of *A* rely on the original signature of *r* which lets them use a separate actual argument; due to polymorphism and dynamic binding, the redefined version of *r* may be used that does not perform any locking, which turns its formal argument *x* into a traitor. Furthermore, the inherited precondition *x.some_property* becomes a correctness condition; this is not what the client expects when calling *a.r (my_x)* in a state where *my_x.some_property* does not hold. Clearly, the covariant redefinition policy is inappropriate here. The strengthened signature conformance rule (8.14.4 /VNCS/) introduced in the Eiffel standard does not solve this problem either.

A straightforward application of Eiffel rules for feature redefinition leads to unsoundness and deadlocks. A more precise definition of the conformance relation between types is necessary; the rules for feature redefinition and the semantics of contracts need to be refined too. The type system proposed in this dissertation (see chapter 6) provides sound rules; the use of polymorphism in combination with other techniques such as precursor calls, contract redefinition, and attached types is discussed in section 9.1.

```
class A
feature
    r (x: separate X)
        require
            x.some_property        -- Wait condition
        do ... end
end

class B
inherit A redefine r end
feature
    r (x: X)
        do
            x.f                    -- x may become a traitor
        end
end

-- in class C
a: A
b: B
my_x: separate X
 ...
a.r (my_x)   -- Valid
a := b       -- Polymorphic assignment
a.r (my_x)   -- Valid but creates a traitor
```

Figure 5.14: Problems with routine redefinition

5.11 Genericity

The impact of concurrency on genericity has not been studied in SCOOP_97. It seems that genericity should not cause any problems in a concurrent context; nevertheless, the absence of consistency rules for generic parameters (similar to rules SC1–SC4 for entities and expressions) leads to potential atomicity violations. A number of issues, in particular related to the combination of genericity and polymorphism, are not addressed by the validity rules of the model:

- *Separateness of formal generic parameters*
  Should separate formal generic parameters be allowed? If yes, how are they specified? Do we need to use constrained genericity, e.g.

  class *A* [*G* −> **separate** *C*]

  or are unconstrained parameters separate by default, i.e. the declaration **class** *B* [*G*] is equivalent to **class** *B* [*G* −> **separate** *ANY*]?

- *Conformance of formal generic parameters in the context of inheritance*
  Should the specialisation of a separate parameter into a non-separate be allowed, i.e.

```
class A [G -> separate C] ... end

class B [G -> C]
inherit A [G]
 ...
end
```

or the other way round, i.e.

```
class A [G -> C] ... end

class B [G -> separate C]
inherit A [G]
 ...
end
```

- *Conformance of actual and formal generic parameters*
  Is it safe to use a separate actual generic parameter where the corresponding formal parameter is non-separate, and vice-versa?
- *Conformance of generic types*
  Is *A* [*X*] a subtype of *A* [**separate** *X*] or vice-versa?

All these topics are discussed in section 9.2; an appropriate extension of the type system is proposed to eliminate the danger of traitors.

5.12 Practical considerations

The theoretical inconsistencies pointed out in the previous sections are sources of unsoundness; they also badly influence the practicality of the model, limiting its expressiveness and making it less convenient to use. This section presents several problems that do not compromise the soundness of SCOOP_97 but nonetheless complicate the practice of concurrent programming.

5.12.1 Enclosing routines

The most common complaint about SCOOP_97 concerns the requirement to wrap all separate calls in a routine that takes their targets as arguments [33, 32, 27]. Enclosing routines are justified if we want to specify some wait conditions or perform a sequence of separate calls in mutual exclusion. On the other hand, wrapping calls in routines may become tedious if we only want to perform a single separate call and we do not need wait conditions: we are still forced to write an additional routine for each call, just for the sake of proper synchronisation through argument passing, as illustrated in figure 5.15. The programmer is interested in calling three features on *my_x* without insisting on the atomicity, i.e. other clients are allowed to interfere between the calls. Three auxiliary routines *r*, *s*, and *t* are necessary to wrap the calls; this puts additional burden on the programmer and makes the code unwieldy.

To solve this problem, Brooke and Paige [32] suggest a “lazy lock” mechanism whereby a single separate call appearing in a context where its target is not locked acquires an implicit lock on the target, so that the programmer may simply write

```
my_x: separate X
 ...
r (my_x)
s (my_x, ''Hello world!'')
t (my_x, 5)
 ...
r (x: separate X)
    do
        x.f
    end

s (x: separate X; s: STRING)
    do
        x.g (s)
    end

t (x: separate X; i: INTEGER)
    do
        x.h (i)
    end
```

Figure 5.15: Burdensome enclosing routines

```
my_x.f; my_x.g (''Hello world!'') ; my_x.h (5)
```

The call *my_x.f* is implicitly converted to *r (x)* with the enclosing routine *r* generated automatically by the compiler, and similarly for other calls. Unfortunately, this solution eliminates the syntactic distinction between a single separate call and a call that is part of a longer atomic sequence; this leads to incorrect assumptions about atomicity (see section 4.2.3). Therefore, we need another approach which distinguishes syntactically between the two cases but minimises the annotation burden while preserving the intended semantics and atomicity guarantees. We present our solution in section 9.3.4.

5.12.2 Deferred classes

SCOOP_97 prohibits the use of more than one keyword among **separate**, **expanded**, and **deferred** for decorating entities or class headers; other authors [43, 60, 34] accept this limitation. Indeed, the use of **expanded** and **separate** together does not make much sense if we consider that an expanded entity denotes an object that "sits" inside another object, rather than a reference; hence there is no reference to cross the boundary of a processor (see section 6.10 for a thorough discussion). On the other hand, there is no good reason to prohibit separate entities of a deferred type. Deferred classes are an important tool of object-oriented design — in particular for class taxonomies and constrained genericity — so their use must not be restricted in a concurrent context. Declarations of the form *x*: **separate** *X*, where *X* is deferred, should be permitted.

```
r (s: separate STRING)
    require
        not s.is_empty
    local
        s2: separate STRING
    do
        s2 := s.twin
        s.append (s2)    -- Invalid call
        s.append (s)     -- Invalid call
    end
```

Figure 5.16: Problems with sequential-to-concurrent reuse

5.12.3 Software reuse

The reuse of sequential libraries is easier in SCOOP_97 than in other popular programming languages, e.g. multithreaded Java and C♯. The code may be easily reused and extended through inheritance; a sequential class can be used in a concurrent application with no need for modifications. For example, the class *BOUNDED_QUEUE* [*G*] may represent a shared bounded buffer:

```
buffer : separate BOUNDED_QUEUE [INTEGER]
```

There is no need to provide a specialised version of the class equipped with additional synchronisation code. Unfortunately, such reuse is hindered by the limitations of the type system. Figure 5.16 illustrates the problem: class *STRING* from the standard EiffelBase library cannot be simply used in the context of routine *r* because it is not "concurrency-aware". Feature *append* in class *STRING* takes a non-separate argument of type *STRING*, whereas the actual argument of the call *s.append* (*s2*) is separate. Following the consistency rule SC3 (4.2.8), the program is rejected although it is perfectly safe: *s* and *s2* are non-separate from each other (*s2* is a clone of *s*). Even more surprisingly, the call *s.append* (*s*) is rejected too. To satisfy the consistency rules, we would need to provide a modified version of *STRING* with a redefined feature *append*. The need to provide a specialised version of a class for each particular use contradicts the very principle of reuse.

The conflict between the wait semantics of separate preconditions and the correctness semantics of other preconditions hinders the reuse of concurrent code in a sequential context. Consider figure 5.17. Feature *store* has a separate precondition; hence, this precondition is a wait condition. If the client calls *store* with a separate actual argument *some_buffer*, the semantics is applied correctly, i.e. the client waits until the precondition is satisfied. On the other hand, if the actual argument is non-separate (recall that SCOOP_97 does not prohibit attachments from non-separate to separate) and the precondition does not hold, the wait semantics still applies, resulting in an immediate deadlock; the state of *my_buffer* will never change because *my_buffer* is handled by the client's processor, and that processor is waiting. As a result, concurrent-to-sequential code reuse may lead to deadlocks.

An expressive type system and a clarified semantics of assertions should enable an unconstrained sequential-to-concurrent and concurrent-to-sequential reuse. See section 10.4 for a detailed discussion of these issues in the proposed framework.

```
store (buffer: separate BUFFER [INTEGER]; v: INTEGER)
    require
        not buffer.is_full
    do
        buffer.put (v)
    end

some_buffer: separate BUFFER [INTEGER]
my_buffer: BUFFER [INTEGER]
...
store (some_buffer, 7)
store (my_buffer, 3)          -- Potential deadlock
```

Figure 5.17: Problems with concurrent-to-sequential reuse

5.13 Discussion

We have pointed out several inconsistencies and limitations of the original model. Although they manifest themselves in many different ways, most problems are a combination of two main issues:

- An incomplete integration of the **separate** mechanism with the rest of the language.
- An unclear semantics of contracts.

Chapters 6 – 10 address all the identified problems, and take the above critique as roadmap for the development of the current SCOOP framework. Chapter 6 clarifies several elements of the computational model, and introduces a type system for safe concurrency. Chapter 7 optimises the access control policy to increase the expressiveness of the model and increase the amount of potential parallelism. Chapter 8 discusses the application of Design by Contract to concurrency: a new generalised semantics of assertions is proposed to take advantage of the modelling power of DbC and to use contracts in correctness proofs of concurrent software. Chapter 9 shows how several advanced object-oriented mechanisms are supported in SCOOP; polymorphism, dynamic binding, feature redefinition, inheritance anomalies, genericity, agents, and once functions are discussed in detail. Finally, chapter 10 comments on the practical aspects of the proposed framework and illustrates its use with a number of examples.

6

Type system for SCOOP

THIS chapter presents an enriched type system for safe synchronisation. A number of issues identified in the previous chapter are addressed here. First, we clarify the semantics of the feature call and the feature application mechanisms, and define the synchronisation rules. Second, we introduce a richer notion of type which supports precise reasoning about object locality. A unified call validity rule is proposed; the type rules are further refined to deal with detachable and expanded types. A new import mechanism for object structures is introduced, and the semantics of object test is refined to support safe downcasts between separate and non-separate types. Finally, we formalise the type system and argue informally its soundness, in the sense that correctly typed programs do not introduce traitors.

6.1 Computational model

Concurrency in SCOOP relies on the basic mechanism of object-oriented computation: the feature call. Each object is handled by a *processor* — a conceptual thread of control — referred to as the object's *handler*. (Throughout the rest of the discussion, we will use these two terms interchangeably.) All features of a given object are executed by its handler. Several objects may have the same handler. The mapping between an object and its handler does not change over time; we do not consider object migration. Objects handled by different processors are called *separate*; objects handled by the same processor are *non-separate*. We take the processor model "as is" from SCOOP_97 (a detailed discussion of processors can be found in section 4.2.1); other elements of the computational model — feature call, mutual exclusion, condition synchronisation — need to be refined. Let us first define the notions of separate and non-separate call; we will often use them in other definitions and rules.

Definition 6.1.1 (Separate call) *A feature call is separate if and only if its target is handled by a different processor than the client object.*

Definition 6.1.2 (Non-separate call) *A feature call is non-separate if and only if its target is handled by the same processor as the client object.*

In contrast to SCOOP_97, the separateness of a call is a *dynamic* property, i.e. it depends on the relative locality of the client object supplier objects at run time, and not on the annotations decorating the entities that represent the objects. A call on a separate entity is not necessarily separate: an entity declared as **separate** might represent a non-separate object at run time (see section 6.2.1).

To capture precisely the semantics of the feature call mechanism, it is necessary to distinguish between a feature *call* and a feature *application*. A feature call is the sequence of actions performed by the client's handler; a feature application is performed by the supplier's handler. We rely on this distinction to define the rules for other essential mechanisms in the following sections and chapters. Definitions 6.1.3 and 6.1.4 below establish the roles and the responsibilities of clients and suppliers. A client's handler is in charge of argument passing, scheduling the feature, and (possibly) waiting for a result; a supplier's handler is in charge of applying the feature after ensuring a proper synchronisation.

Every processor has a *call stack* and a *request queue*. The call stack is used for handling non-separate calls in exactly the same way as in a sequential system. The request queue keeps feature calls that target objects handled by the current processor but have been issued by another processor. These requests are serviced in the order of their arrival; hence the FIFO scheduling of feature calls within a processor. A processor may be in one of two states: *locked* or *unlocked*. The *locked* state means that the processor is under the control of another processor and is ready to receive feature call requests from that processor. The *unlocked* state means that the processor is not ready to receive any requests. Each processor performs repeatedly the following actions:

- Request processing: if there is an item on the call stack, pop the item from the stack and process it, i.e.
  - (Feature application) if the item is a feature request, apply the feature;
  - (Unlocking) if the item is an $unlock$ operation, set your state to *unlocked*.
- Request scheduling: if the call stack is empty but the request queue is not empty, dequeue an item and push it onto the stack.
- Idle wait: if both the stack and the queue are empty, wait for new requests to be enqueued.

Throughout the rest of this chapter, we will represent the request queues as sequences (read from left to right) of feature calls wrapped in square brackets, e.g.

$$P_x : \quad [x.f][y.g][x.h(5)][unlock]$$

means that the processor P_x has three feature calls and an unlocking operation in its queue; they will be serviced in the order: $x.f$, then $y.g$, then $x.h\ (5)$, then $unlock$. (Note that no requests may appear in the queue after an $[unlock]$.) An empty request queue is represented as

$$P_x : \quad -$$

6.1.1 Feature call

What happens when the feature call $x.f\ (a)$ is performed? First, argument passing takes place, i.e. the formal arguments of f are bound to the corresponding actual arguments a. Second, the target's handler is asked to execute f. There are two possibilities here: if the call is non-separate, the feature is scheduled for an immediate execution. (This is achieved just like in a sequential setting, i.e. the current execution state is saved in a stack frame and pushed on top of the current processors's execution stack, and the control is passed to the called feature.) If the call is separate, the feature is scheduled to execute as soon as all the previous calls on the target's

handler have terminated. (This is achieved by appending a feature request to the request queue of the target processor.) Finally, if the feature *f* is a command, the call is already complete; if it is a query, the client's handler waits for its result (*wait by necessity* applies).

Definition 6.1.3 (Feature call semantics) *Call $x.f\ (\overline{a})$ results in the following sequence of actions performed by the client's handler P_c:*

1. *Argument passing: bind the arguments of f to the corresponding actual arguments $\overline{a}$.*
2. *Feature request: ask x's handler P_x to apply f to x. There are two cases here:*
   (a) *If the feature call is non-separate, i.e. $P_c = P_x$, f is scheduled for an immediate execution.*
   (b) *If the feature call is separate, i.e. $P_c \neq P_x$, f is scheduled to execute after the previous calls on P_x.*
3. *Wait by necessity: if f is a query, wait for its result.*

In the following example that illustrates the possible combinations of separate and non-separate calls with and without wait by necessity, the client object (**Current**) is handled by the processor P_c, and the object attached to *x* is handled by P_x.

```
my_x: X
i: INTEGER

r (x: separate X)
    do
        my_x.f (5)        -- non-separate, no wait by necessity
        i := my_x.g       -- non-separate, wait by necessity
        x.f (10)          -- separate, no wait by necessity
        i := x.g          -- separate, wait by necessity
    end
```

Assuming that the request queue of P_c is empty when the execution of *r* starts (the request queue of P_x must be empty), the following states of the request queues may be observed:

- (Command call *my_x.f* (5)) The call is non-separate because *my_x* is handled by P_c; the current execution state is saved on P_c's call stack, and *my_x.f* (5) is executed synchronously. (Only after the execution of *my_x.f* (5) will P_c proceed to the next instruction in the body of *r*.) The request queues are not involved in this operation, hence
  $P_c: -$ $P_x: -$
- (Query call *my_x.g*) Similar to the previous call. Wait by necessity is "transparent" here because the call is synchronous anyway. The request queues are not involved in this operation, hence
  $P_c: -$ $P_x: -$
- (Command call $x.f$ (10)) The call is separate because *x* is handled by P_x and $P_x \neq P_c$. A feature request $[x.f(10)]$ is added to P_x's queue. The call is asynchronous because wait by necessity does not apply; P_c moves immediately to the next operation.
  $P_c: -$ $P_x: [x.f(10)]$

- (Query call $x.g$) Similar to the previous call but wait by necessity applies, i.e. P_c waits until $x.g$ has terminated and returned a result. This gives an impression of synchrony. If P_x has not dequeued the previous request yet, the queues look like this:
  $P_c: \; - \qquad P_x: \; [x.f(10)][x.g]$
  If P_x has already dequeued the previous request, the queues look like that:
  $P_c: \; - \qquad P_x: \; [x.g]$

It is important to notice that request queues are only involved in handling the requests issued by other processors. That is why P_c's queue in the above example remains empty throughout the execution of the whole routine body, even when non-separate calls (where P_c is also the supplier's handler) are performed.

6.1.2 Feature application

The application of a feature is performed by the supplier's handler. The application of f on x requires a correct synchronisation: f may only be executed when its formal arguments are reserved on behalf of the supplier, i.e. their handlers are locked for the exclusive use of x's handler, and the precondition of f holds. If either of these two conditions does not hold, the supplier's handler blocks until both conditions are satisfied.

Definition 6.1.4 (Feature application semantics) *The application of feature f on target x, requested by client c, results in the following sequence of actions performed by the supplier's handler P_x:*

1. *Synchronisation: wait until the formal arguments of f are reserved, i.e. their handlers are locked on behalf of P_x, and the precondition of f holds.*
2. *Execution: if f is a routine, execute its body; if f is an attribute, evaluate the corresponding field.*
3. *Result: if f is a query, return its result to c.*
4. *Release: ask each handler locked in the synchronisation step to unlock itself when it terminates.*

The phrasing of the last step is very important. The release of locks is asynchronous, i.e. the processor executing f does not wait for all the processors locked by f to terminate the execution of features requested in f's body. Instead, it appends an $[unlock]$ request to the request queue of each locked processor. As a result, different processors may become unlocked at different times. The executor of f does not wait for the unlocking; it continues its execution. For example, if the body of f contains the following sequence of calls on targets x and y handled by processors P_x and P_y locked in the first step:

```
f (x, y: separate X)
    do
        x.g
        x.h
        y.g
        x.h
    end
```

then, after the release step, the request queues of P_x and P_y will look like this:

$$P_x : \ [x.g][x.h][x.h][unlock] \qquad\qquad P_y : \ [y.g][unlock]$$

provided that P_x and P_y have not dequeued any requests in the meantime. This approach enables more parallelism than the CCR-like synchronisation mechanism in SCOOP_97 (whereby both the locking and the unlocking operations are atomic). Ultimately, it permits safe realisation of certain synchronisation scenarios that lead to deadlocks in SCOOP_97.

6.1.3 Synchronisation

SCOOP supports two kinds of synchronisation: mutual exclusion and condition synchronisation. Both of them are ensured by the feature application mechanism described above, combined with the call validity rule 6.5.3 defined later in this chapter. We do not distinguish between separate and non-separate arguments, nor between separate and non-separate preconditions. If an argument is non-separate then it is trivially reserved; all precondition clauses have the wait-semantics. The following rule captures both synchronisation requirements at the time of feature application.

Definition 6.1.5 (Feature application rule) *Before a feature is applied, its formal arguments must be reserved by the supplier's handler, and its precondition must hold.*

Contrary to SCOOP_97, the traditional semantics of argument passing — understood as the binding of formal arguments to the provided actual arguments, performed by the client's handler — is preserved in SCOOP. Although the synchronisation relies on the information about the formal arguments, it is enforced on the supplier's side and does not alter the meaning of the argument passing itself.

The FIFO scheduling implied by the feature call semantics and the atomicity guarantees provided by the feature application mechanism are essential for reasoning about concurrent programs. Coupled with the refined call validity rule, they guarantee that a sequence of calls

```
...
x.f
x.g (a)
x.h
...
```

is executed atomically in the order of their appearance in the program text; no other client may access *x* in the meantime. Section 6.5 discusses this topic in more detail.

Besides ensuring the atomicity of routine bodies, the feature application mechanism provides a simple and convenient way to lock several resources at once: a routine taking several arguments locks *atomically* all the processors handling the arguments. For example, the execution of

```
eat ( left_fork , right_fork )
```

blocks until both arguments have been locked. There is no limit on the number of formal arguments, hence no limit on the number of locks acquired atomically. The scheduler performs

locking in such a way that requests with overlapping sets of requested processors do not deadlock. Also, fairness is guaranteed: no request can be overtaken by another request with the same (or larger) set of requested processors; this means that "equal" requests are scheduled in a FIFO order. (Remarkably, the above call to *eat* expresses all the synchronisation necessary to solve the dining philosophers problem in SCOOP [94]; this is the simplest symmetric solution in the literature.)

Similarly to SCOOP_97, *wait by necessity* maximises the amount of asynchrony: clients only need to wait for the result of query calls (function or argument evaluation); commands (procedures) are executed asynchronously (see definition 6.1.3). In SCOOP, however, wait by necessity does not apply to assertions; postconditions, check instructions, and loop assertions are evaluated asynchronously (see chapter 8).

The presented synchronisation mechanism is refined in chapter 7: section 7.1 relaxes the feature application rule 6.1.5 to account for detachable and attached types of formal arguments; the semantics of argument passing is refined in section 7.2 to enable lock passing between a client and a supplier. Chapter 8 discusses the interplay of feature calls, synchronisation, and DbC.

6.2 From consistency rules to a type system

The rest of this chapter extends Eiffel's type system to capture precisely the *object locality*, i.e. the relative separateness of objects, and to provide type rules which clarify, refine, and formalise the validity and consistency rules of the model. SCOOP_97 attempted to capture the conformance relation between non-separate and separate entities using a set of informal rules; these rules proved insufficient to ensure the intended atomicity and safety guarantees. Additionally, they were too imprecise, which limited the expressiveness of the model (see section 5.4). We take a different approach and rely on formal type rules to ensure the required safety properties. Our type system is backward-compatible with Eiffel's, i.e. correct Eiffel programs do type-check in SCOOP as well.

The type system is fully formalised in section 6.11 but individual rules appear earlier in the chapter to illustrate the discussed topics. If some technical details of a rule, e.g. auxiliary functions, are not clear when it first appears in the text, please consult the formalisation section.

6.2.1 SCOOP types

As pointed out in section 5.1, there is a mismatch between the strict semantics of **separate** annotations requiring that the objects referenced by separate entities be handled by a different processor, and the intuitive non-strict semantics that allows such possibility but does not enforce it, i.e. separate entities *may* represent separate objects or, expressed differently, separate entities denote *potentially separate* objects. This mismatch led to numerous complications and inconsistencies in SCOOP_97: some rules and definitions followed the strict semantics; the others used the non-strict one. We opt for the non-strict semantics of **separate** in our framework because it is the only sound and practical solution; it is enforceable statically and compatible with the modularity requirements.

The **separate** keyword, even with a clear semantics, is not sufficient because it only expresses the separateness of an entity from **Current**; the relative locality of an entity with respect

to another one cannot be captured and reasoned about. Such properties should be expressible in types, in a similar way as object ownership is captured by *ownership types*. (A rich literature on ownership types demonstrates the practicability of the approach; the type system described here was directly inspired by the work of Dietl et al. [49].) Eiffel's type system, based on class types, is too weak; hence the need to introduce a second type component — *processor tag* — which captures the locality of objects. Anticipating the need to accommodate another important mechanism — *attached types* — we add a third type component. Therefore, SCOOP types are represented as triples

$$T = (\gamma, \alpha, C)$$

with the following components:

- *Detachable tag* $\gamma \in \{!, ?\}$
  A type is either attached ($\gamma = !$) or detachable ($\gamma = ?$), in the standard Eiffel sense: entities of an attached type are statically guaranteed to be non-void at run time; detachable entities may be void [96, 53].

- *Processor tag* $\alpha \in \{\bullet, \top, p, \bot\}$
  The processor tag captures the locality of objects represented by an entity of type T, i.e. their separateness or non-separateness with respect to other objects. An entity is either non-separate (handled by the current processor, $\alpha = \bullet$), separate (handled by some processor, $\alpha = \top$), or handled by processor p ($\alpha = p$). '$\bot$' denotes *no processor*; it is used to type **Void**.

- *Class type* C
  This is the "traditional" Eiffel class type. We ignore genericity here; section 9.2 extends the type system with the support for generically derived types.

Since the semantics of a feature call depends on the type of its target — in particular on its processor tag — the software text must indicate it unambiguously. An entity may now be declared as one of:

- *x*: *X*
  x has the type $(!, \bullet, X)$, i.e. objects attached to *x* are non-separate from **Current**.

- *x*: **separate** *X*
  x has the type $(!, \top, X)$, i.e. objects attached to *x* are potentially separate from **Current**.

- *x*: **separate** <*p*> *X*
  x has the type $(!, p, X)$, i.e. objects attached to *x* are handled by the processor known as *p*; they are potentially separate from **Current** but non-separate from other objects handled by *p*.

A type annotation may also include the '?' sign, e.g. *x*: ?**separate** *X*, which sets the detachable tag to '?'. Entities not decorated with '?' are attached, i.e. their detachable tag is '!'.

This syntax minimises the annotation burden (in practice, most entities are non-separate and attached thus require no additional annotations) and ensures the backward-compatibility with sequential Eiffel. This is an important property, given the number of existing libraries. SCOOP syntax may be seen as a straightforward extension of SCOOP_97's notation; the keyword **separate**, however, loses its special status and becomes a mere type annotation.

6.2.2 Processor tags

As pointed above, the distinction between separate and non-separate entities is not sufficient in practice because the relative locality of two separate objects cannot be captured and reasoned about. Explicit processor tags overcome this limitation and support precise reasoning about object locality. Entities declared with the same processor tag, e.g.

```
x: separate <px> X
y: separate <px> Y
z: ?separate <px> Z
```

represent objects handled by the same processor known as *px*. These objects may or may not be separate from the current object; the exact identity of *px* is not known at compile-time, hence the use of the **separate** keyword. But they are certainly not separate from each other. The type system takes advantage of this information to support safe attachments and feature calls between *x*, *y*, and *z* without the need for locking (see section 6.3). The tag *px* must be declared in the class where it is used, or inherited from an ancestor; an attribute-like form

```
px: PROCESSOR
```

is used. *PROCESSOR* is a reserved class name; it may only appear in tag declarations. Processor tags must be unique: no other tag, entity, or feature may be called *px* within the same class. The use of *px* is limited to type annotations. For different objects (also different instances of the same class) *px* may denote a different processor.

In addition to *unqualified* processor tags like *px*, a *qualified* form may be used; such tags are derived from the names of other entities using the call-like notation ".**handler**":

```
r (x: separate X)
    local
        y: separate <x.handler> Y
    do ... end
```

Entity *y* is declared as handled by *x*'s processor. Qualified tags are subject to the same syntactic restrictions as unqualified tags. Additionally, they must not be based on an entity whose object may change or become undefined; otherwise, the type rules would be unsound. Therefore, such tags may only be defined on attached non-writable entities. To prevent chains of unqualified tags, e.g. *y* deriving its processor tag from *x*, *z* from *y* and so on, qualified tags may only be based on entities without an explicit qualified tag. This restriction also eliminates circular dependencies of tags. Rules 6.2.1 and 6.2.2 summarise the validity requirements for processor tags.

Definition 6.2.1 (Valid unqualified processor tag) *An unqualified processor tag p is valid in class C if and only if p is declared as*

```
p: PROCESSOR
```

in C or in one of its ancestors.

Definition 6.2.2 (Valid qualified processor tag) *A qualified processor tag e.**handler** is valid if and only if e is a non-writable entity — a formal argument of the enclosing routine, a constant, or a once function — of an attached type, and e itself has no explicit qualified processor tag.*

Eiffel users may spot a similarity between qualified processor tags and the *anchored declaration* mechanism [53], whereby an entity declared as

```
z: like x
```

takes x's class type; x must be the final name of a feature in the enclosing class. An anchored declaration does not influence the detachability or the separateness of the decorated entity. A qualified processor tag, on the other hand, influences the separateness but not the detachability or the class type of the entity.

6.2.3 Implicit types

A non-writable attached entity e may itself be viewed as having an implicitly declared qualified processor tag e.**handler**. For example, the entities in the following code excerpt

```
r (x: separate X; y: ?Y)
    local
        z: separate Z
    do
        ...
    end

s: STRING = ''I'm a constant''

u: separate U once ... end
```

have the following types:

- x has the declared type $(!, \top, X)$ and an implicit type $(!, x.\mathbf{handler}, X)$
- s has the declared type $(!, \bullet, STRING)$ and an implicit type $(!, s.\mathbf{handler}, STRING)$
- u has the declared type $(!, \top, U)$ and an implicit type $(!, u.\mathbf{handler}, U)$
- y has the declared type $(?, \bullet, Y)$
- z has the declared type $(!, \top, Z)$

Entities y and z do not have an implicit type: y.**handler** makes no sense because y is detachable and may be void; z is writable, so z.**handler** may vary between two consecutive evaluations of z.

Definition 6.2.3 (Implicit type rule) *An attached non-writable entity e of type $T_e = (!, \alpha, C)$ also has an implicit type $T_{e_{imp}} = (!, e.\mathbf{handler}, C)$.*

The corresponding formal rule **T-Implicit** appears in our type system (see section 6.11).

$$\frac{\Gamma \vdash e : (!, \alpha, C), \quad \Gamma \vdash \neg isWritable(e)}{\Gamma \vdash e : (!, e.\mathbf{handler}, C)} \qquad \text{(T-Implicit)}$$

Implicit types of non-writable entities are useful for several purposes; most importantly, they simplify the call validity rules (see section 6.5) and support a seamless reuse of sequential libraries (see section 10.4).

A note on *Current*

The entity **Current** is always attached and non-separate. These properties are captured by the following rule.

Definition 6.2.4 (Type of *Current*) *In the context of class C, **Current** has the type* $(!, \bullet, C)$.

There is no axiom for **Current** among the formal type rules; its type is always carried in the typing environment Γ, just like the types of local variables and formal arguments. Rules 6.2.2, 6.2.3, and 6.2.4 clarify the meaning of the processor tag '•' and type annotations like

x: X

Since **Current** is both attached and non-writable, **Current.handler** is a valid processor tag; therefore, a non-separate entity may also be declared as

x: **separate** <**Current.handler**> X

By definition 6.2.4, the processor tag of **Current** is '•'. Therefore, '•' and **Current.handler** are equivalent, so that the two declarations above have exactly the same meaning and may be used interchangeably. In practice, it is easier to use the shorter form x: X which is immediately understandable and less burdensome.

A note on *Void*

Void denotes a detached reference, i.e. a reference pointing to no object. Therefore, **Void** must have a detachable type. Its class type, just like in sequential Eiffel, is *NONE*. Furthermore, since it represents no object, it has no handler. These properties are captured by the following rule.

Definition 6.2.5 (Type of *Void*) ***Void** has the type* $(?, \bot, NONE)$ *in all contexts.*

This is expressed more formally by the following axiom of the type system.

$$\frac{}{\Gamma \vdash \textbf{Void} : (?, \bot, NONE)} \quad \text{(T-Void)}$$

6.3 Subtyping

The subtype relation on SCOOP types is based on the conformance of individual type components. The conformance of class types is based on subclassing: a class *D* conforms to *C*, i.e. $D \sqsubseteq C$, if and only if *D* is a descendant of *C*. In particular, we have $C \sqsubseteq C$, $C \sqsubseteq ANY$, $NONE \sqsubseteq C$, and $E \sqsubseteq D \wedge D \sqsubseteq C \implies E \sqsubseteq C$. Precise rules for subclassing are given in figure 6.17. Processor tags are ordered in a lattice, with the top element '$\top$' and the bottom element '$\bot$'; other tags conform to '$\top$' but not to each other, as illustrated in figure 6.1. Finally, the detachable tag '!' conforms to itself and '?'; the latter only conforms to itself. The resulting subtype relation is captured by a set of rules in figure 6.2.

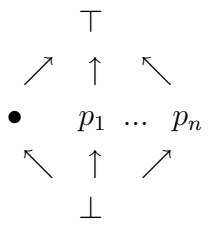


Figure 6.1: Conformance of processor tags

$$\frac{\Gamma \vdash C \sqsubseteq C'}{\Gamma \vdash (\gamma, \alpha, C) \preceq (\gamma, \alpha, C')} \quad \text{(S-Subclass)}$$

$$\frac{\Gamma \vdash C \sqsubseteq C'}{\Gamma \vdash (\gamma, \alpha, C) \preceq (\gamma, \top, C')} \quad \text{(S-Top)}$$

$$\frac{\Gamma \vdash C \sqsubseteq C'}{\Gamma \vdash (\gamma, \bot, C) \preceq (\gamma, \alpha, C')} \quad \text{(S-Bottom)}$$

$$\frac{\Gamma \vdash (?, \alpha, C) \preceq (?, \beta, C')}{\Gamma \vdash (!, \alpha, C) \preceq (?, \beta, C')} \quad \text{(S-Attached)}$$

Figure 6.2: Subtyping rules

Let's see how subtyping works in practice. Assuming that class *Y* inherits from *X*, i.e. $Y \sqsubseteq X$, the type $T_2 = (!, \bullet, Y)$ is a subtype of $T_1 = (?, \top, X)$ because one can construct a derivation using the rules **S-Attached** and **S-Top**:

$$\dfrac{\dfrac{\Gamma \vdash Y \sqsubseteq X}{\Gamma \vdash (?, \bullet, Y) \preceq (?, \top, X)}\ \text{(S-Top)}}{\Gamma \vdash (!, \bullet, Y) \preceq (?, \top, X)}\ \text{(S-Attached)}$$

Similarly, $T_3 = (!, \top, X)$ is a subtype of T_1; the derivation uses **S-Attached** and **S-Subclass**:

$$\dfrac{\dfrac{\Gamma \vdash X \sqsubseteq X}{\Gamma \vdash (?, \top, X) \preceq (?, \top, X)}\ \text{(S-Subclass)}}{\Gamma \vdash (!, \top, X) \preceq (?, \top, X)}\ \text{(S-Attached)}$$

The type rules for assignment, feature call, and object creation use the subtyping relation to ensure the the soundness of attachments. All the rules are given in figures 6.22 and 6.23 but we will be introducing individual rules in the rest of this section as the discussion progresses. (If some technical details of a rule, e.g. auxiliary functions, are not clear when it first appears in the text, please consult the formalisation section 6.11.) We start with assignments. Consider the following code excerpt.

```
x: ?separate X
y: Y
z: separate X
my_z: X
 ...
x := y
x := z
y := x          -- Invalid
z := x          -- Invalid
my_z := z       -- Invalid
```

We use the standard type rule for assignments, which requires the source to conform to the target. The assignments $x := y$ and $x := z$ are valid because x has the type T_1, y has the type T_2, z has the type T_3, and the above derivations show the conformance of T_2 and T_3 to T_1. The assignments $y := x$, $z := x$, and $my\_z := z$ are invalid because the types do not conform. In the case of $y := x$ class types are not compatible; $z := x$ assigns from detachable to attached. Finally, $my\_y := z$ is an assignment from separate to non-separate. Note that the latter is prohibited in SCOOP_97 by an explicit consistency rule SC1 (4.2.6); our type system eliminates the need for such special rules. The formal type rule **T-Assign** given below states that an assignment is valid if and only if its target is writable, and the type of the source conforms to the type of the target.

$$\frac{\Gamma \vdash isWritable(x),\ \ \Gamma \vdash x : T_x,\ \ \Gamma \vdash e : T_e,\ \ \Gamma \vdash T_e \preceq T_x}{\Gamma \vdash x := e \diamond} \qquad \text{(T-Assign)}$$

The actual arguments of a feature call must conform to the corresponding formal arguments of the called routine. Consider the following example:

```
a: separate X
b: X
c: ?X
f (x, y: separate X)    do ... end
g (x: X)      do ... end
h (x: ?X): separate <p> X  do ... end
 ...
f (a, b)
f (a, c)        -- Invalid
g (a)           -- Invalid
a := h (b)
a := h (a)      -- Invalid
```

The call $f\ (a,\ b)$ is valid because the actual arguments a and b conform to the corresponding formals x and y: $(!, \top, X) \preceq (!, \top, X)$, and $(!, \bullet, X) \preceq (!, \top, X)$. The call $f\ (a,\ c)$ is invalid because c does not conform to the corresponding formal: $(?, \bullet, X) \not\preceq (!, \top, X)$. Similarly, the call $g\ (a)$ is invalid because $(!, \top, X) \not\preceq (!, \bullet, X)$. (Here too, SCOOP_97 required an explicit rule SC1 to prohibit such attachments.) The statement $a := h\ (b)$ is valid because b conforms to the formal argument of h, i.e. $(!, \bullet, X) \preceq (!, \bullet, X)$, and because the assignment satisfies the rule **T-Assign**. On the other hand, a does not conform to the formal argument of h, hence the

statement $a := h\ (a)$ is invalid.

Rules **T-CCallUnqual**, **T-QACallUnqual**, and **T-QFCallUnqual** capture the correctness of unqualified calls. (The auxiliary function *FeatureType* yields the type of a feature in a given class; see section 6.11 for details.) Both of them require the conformance of actual argument types to the corresponding formal argument types; additionally, **T-QACallUnqual** and **T-QFCallUnqual** say that the result of the call has the same type as the result type of the called feature.

$$\frac{\begin{array}{l}\Gamma \vdash \mathbf{Current} : (!, \bullet, C) \\ \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow \varnothing \quad n \geq 0 \\ \Gamma \vdash a_i : T_i' \quad i \in 1..n, \quad \bar{a} = a_1..a_n, \quad \Gamma \vdash \forall_{i \in 1..n}\ T_i' \preceq T_i\end{array}}{\Gamma \vdash f(\bar{a})\ \diamond} \qquad \text{(T-CCallUnqual)}$$

$$\frac{\Gamma \vdash \mathbf{Current} : (!, \bullet, C), \quad \Gamma \vdash FeatureType(C, f) = T}{\Gamma \vdash f : T} \qquad \text{(T-QACallUnqual)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathbf{Current} : (!, \bullet, C) \\ \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow T \quad n \geq 0 \\ \Gamma \vdash a_i : T_i' \quad i \in 1..n, \quad \bar{a} = a_1..a_n, \quad \Gamma \vdash \forall_{i \in 1..n}\ T_i' \preceq T_i\end{array}}{\Gamma \vdash f(\bar{a}) : T} \qquad \text{(T-QFCallUnqual)}$$

The above rules are not strong enough to handle qualified calls. The next section introduces two type combinators required for the type-checking of qualified calls, and defines the corresponding type rules.

6.4 Type combinators

Processor tags are always relative to the current object. An entity declared as non-separate, e.g. a: A, is seen as such by the current object (and by its non-separate clients). Separate clients, however, should see a as separate, because from their point of view it is handled by a different processor. Figure 6.3 illustrates this problem. For *o1*, the field a has the type $(!, \bullet, A)$; it is attached to *o2* which is indeed a non-separate object. The field b of *o2* also has a non-separate type $(?, \bullet, B)$ and it represents a non-separate object *o3*. The three objects are non-separate from each other; therefore, *o1* should see *o3* as non-separate, i.e. $a.b$ should have the type $(?, \bullet, B)$. For *o4*, however, both *o2* and *o3* are separate; the corresponding references must be separate too. That is why a is declared as separate in *o4*; the expressions $x.a$ and $x.a.b$ evaluated in the context of *o4* have the types $(!, \top, A)$ and $(?, \top, B)$ respectively. Similarly, the expression $y.b$ should be of type $(!, \top, B)$.

Intuitively, following a non-separate reference preserves the locality of a type (its processor tag); following a separate reference turns the type into a separate one. We use these informal rules to derive the type combinator '$\star$' for result types (see figure 6.4). It is used to calculate the type T_e of a query call $x.f\ (...)$ from the type T_{target} of the target x and the result type T_{result} of f:

$$T_e = T_{target} \star T_{result}$$

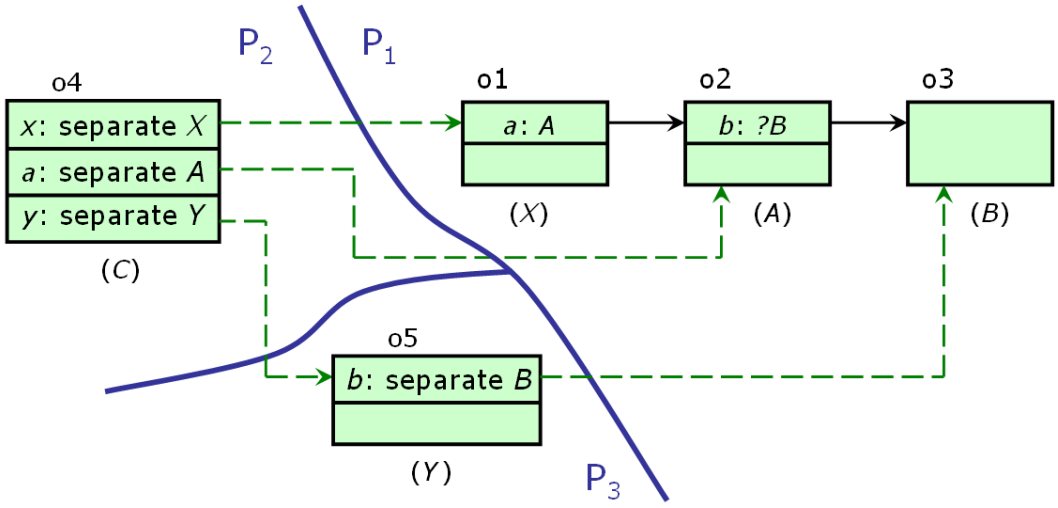


Figure 6.3: Relative separateness of objects

(Result type) $\star : Type \times Type \longrightarrow Type$

$$(\gamma, \alpha, C) \star (\lambda, \beta, D) = \begin{cases} (\lambda, \alpha, D) & \text{if } \beta \in \{\bullet, \mathbf{Current.handler}\} \\ (\lambda, \top, D) & \text{otherwise} \end{cases}$$

(Argument type) $\otimes : Type \times Type \longrightarrow Type$

$$(\gamma, \alpha, C) \otimes (\lambda, \beta, D) = \begin{cases} (\lambda, \alpha, D) & \text{if } \beta \in \{\bullet, \mathbf{Current.handler}\} \wedge \alpha \neq \top \\ (\lambda, \top, D) & \text{if } \beta = \top \\ (\lambda, \bot, D) & \text{otherwise} \end{cases}$$

Figure 6.4: Type combinators

Type combination with '$\star$' should be understood as follows. The resulting type takes the detachable tag and the class type from the second type. If both combined types are non-separate, then the resulting type is non-separate as well ('$\bullet$'). If the second type is non-separate then the resulting type has the processor tag of the first type. Otherwise, the resulting type is separate ('$\top$').

For example, the expression $x.a$ evaluated by *o4* in figure 6.3 has the type $(!, \top, A)$ because x has the type $(!, \top, X)$, a has the type $(!, \bullet, A)$, and $(!, \top, X) \star (!, \bullet, A) = (!, \top, A)$. The expression $a.b$ evaluated by *o1* has the type $(?, \bullet, B)$ because a has the type $(!, \bullet, A)$, b has the type $(?, \bullet, B)$, and $(!, \bullet, A) \star (?, \bullet, B) = (?, \bullet, B)$. The expression $x.y.z$ in the following code excerpt

```
-- in class C
r (x: separate <px> X)
    do
        my_x := x.y.z
    end
```

```
-- in class X
y: Y

-- in class Y
z: ?Z
```

has the type $(?, px, Z)$; the reference x is separate but the precise information about its locality captured by the processor tag *px* is preserved by the type combination:

$$(!, px, X) \star (!, \bullet, Y) \star (?, \bullet, Z) = (!, px, Y) \star (?, \bullet, Z) = (?, px, Z)$$

A similar combinator is needed for the types of formal arguments because they are also defined with respect to the current object; other clients may see them differently. In the following scenario:

```
-- in class X
a: A
f (fa: ?A) do ... end

-- in class C
a: A
my_x: X
r (x: separate X)
    do
        x.f (a) -- Invalid
    end
...
my_x.f (a)
```

the routine *f* in class *X* expects a non-separate argument. The call *my_x.f (a)* is valid because the type of the actual argument *a* conforms to the type of the formal *fa*, i.e. $(!, \bullet, A) \preceq (?, \bullet, A)$. However, this is also true of the call $x.f\ (a)$ which should be prohibited because it turns *fa* into a traitor! The fact that the target of this call is separate should be reflected in the type of *fa* seen by the client, so that the latter must provide an actual argument that is non-separate *from the target*, and not from the client itself. This is enforced by the type combinator '$\otimes$' for argument types (see figure 6.4). The type T_{actual}, to which the actual argument a of the call $x.f\ (a)$ must conform, is derived from the type T_{target} of the target x and the type T_{formal} of the formal argument of f:

$$T_{actual} = T_{target} \otimes T_{formal}$$

'$\otimes$' follows a pattern similar to '$\star$'; however, it may only preserve a type or restrict it, i.e. set the processor tag to '$\bot$'. In the above example, the type of actual argument expected by the call to $x.f$ is $(?, x.\mathbf{handler}, A)$, because (by the rule 6.2.3) x has the type $(!, x.\mathbf{handler}, X)$, and $(!, x.\mathbf{handler}, X) \otimes (?, \bullet, A) = (?, x.\mathbf{handler}, A)$. Therefore, the call $x.f\ (a)$ is invalid because $(!, \bullet, A) \not\preceq (?, x.\mathbf{handler}, A)$. In SCOOP_97, this call would be invalid because of the consistency rule SC2 (4.2.7); in fact, the rule SC2 would prohibit any calls to $x.f$ because f takes a non-separate argument. This is clearly too restrictive, e.g. a call $x.f\ (x.a)$ should be valid because we know that $x.a$ is non-separate from x. Unlike SCOOP_97, our type system supports such calls, as long as the actual argument conforms to the expected type; $x.a$ does:

$$\begin{aligned}(!, x.\mathbf{handler}, X) \star (!, \bullet, A) &\preceq (!, x.\mathbf{handler}, X) \otimes (?, \bullet, A) \\ (!, x.\mathbf{handler}, A) &\preceq (?, x.\mathbf{handler}, A)\end{aligned}$$

Alternatively, the client may use an explicit processor tag to capture the non-separateness of some entity with respect to x. An unqualified tag may be used, e.g.

```
−− in class C
px: PROCESSOR
b: separate <px> A
r (x: separate <px> X)
    do
        x.f (b)
    end
```

Here, the expected type of the argument is $(?, px, A)$ because $(!, px, X) \otimes (?, \bullet, A) = (?, px, A)$. The actual argument *b* has a conforming type $(!, px, A)$, therefore the call is valid. Of course, declaring the formal argument of *r* with an explicit processor tag *px* forces all the calls to *r* to use an actual argument handled by *px*. A solution based on a qualified processor tag does not suffer from this limitation:

```
−− in class C
r (x: separate X)
    local
        b: separate <x.handler> A
    do
        b := ...
        x.f (b)
    end
```

Here, the expected argument type is $(?, x.\mathbf{handler}, A)$ because $(!, x.\mathbf{handler}, X) \otimes (?, \bullet, A) = (?, x.\mathbf{handler}, A)$. The actual argument *b* has the conforming type $(!, x.\mathbf{handler}, A)$, hence the call $x.f\ (b)$ is valid. The use of a qualified processor tag is allowed because *x* is attached and non-writable, as required by the rule 6.2.2.

The type rules **T-CCallQual**, **T-QACallQual**, and **T-QFCallQual** below use both type combinators '$\star$' and '$\otimes$' to ensure the soundness of qualified calls. They require the conformance of actual argument types to the corresponding formal argument types combined, using '$\otimes$', with the target type; additionally, **T-QACallQual** and **T-QFCallQual** say that the result of a call has the same type as the result type of the called feature, combined with the target type using '$\star$'.

$$\frac{\begin{array}{l}\Gamma \vdash e : T_e,\ \ \Gamma \vdash isControlled(T_e),\ \ \Gamma \vdash ClassType(T_e) = C \\ \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow \varnothing \quad n \geq 0 \\ \Gamma \vdash a_i : T'_i \quad i \in 1..n, \quad \bar{a} = a_1..a_n, \ \ \Gamma \vdash \forall_{i \in 1..n}\ T'_i \preceq T_e \otimes T_i\end{array}}{\Gamma \vdash e.f(\bar{a})\ \diamond} \qquad \text{(T-CCallQual)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : T_e,\ \ \Gamma \vdash isControlled(T_e),\ \ \Gamma \vdash ClassType(T_e) = C \\ \Gamma \vdash FeatureType(C, f) = T\end{array}}{\Gamma \vdash e.f : T_e \star T} \qquad \text{(T-QACallQual)}$$

$$\frac{\Gamma \vdash e : T_e,\ \ \Gamma \vdash isControlled(T_e),\ \ \Gamma \vdash ClassType(T_e) = C \quad \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow T \quad n \geq 0 \quad \Gamma \vdash a_i : T'_i \quad i \in 1..n,\ \ \bar{a} = a_1..a_n,\ \ \Gamma \vdash \forall_{i \in 1..n}\ T'_i \preceq T_e \otimes T_i}{\Gamma \vdash e.f(\bar{a}) : T_e \star T} \quad \text{(T-QFCallQual)}$$

Discussion

As will be shown in subsequent sections, the type rules discussed so far ensure the absence of traitors. Due to the required type conformance between the source and the target of an attachment (assignment or argument passing) there is no possibility of attaching an entity to an object handled by a different processor than the declared processor of the entity. This is essential for the safety of concurrent computations. Remarkably, we use traditional type rules to check the correctness of assignments and feature calls; there is nothing "concurrent" about them. The only noticeable modification is the presence of type combinators and an auxiliary function *isControlled*. The next section addresses the controllability issue and completes the discussion of call validity.

```
-- in class C
r (x: separate X)
    do
        x.f (x.y.a)
        x.y.z.f (x.a)
        x.y.f (x.y.z.y.z.a)
         ...
    end

-- in class X
y, z: X
a: A
f (b: ?A) do ... end
```

Figure 6.5: Multi-dot separate expressions

Our enriched type system eliminates the need for explicit consistency rules SC1, SC2, and SC3 of SCOOP_97 which, as demonstrated in section 5.4, are unsound and too restrictive. There are multiple benefits: on one hand, the treatment of separate and non-separate entities is unified, which simplifies the programming practice; on the other hand many convenient constructs — such as multi-dot expressions on separate targets, or separate calls to features taking non-separate arguments — are now freely usable. The example in figure 6.5 is valid in SCOOP but not in SCOOP_97 because the latter does not permit any multi-dot expressions on x and calls to f in the body of r, although they can be perfectly safe. If one considers the class X to be a "sequential" library class, the reuse of X in a concurrent context is now possible; this was not the case before (see section 5.12.3). Section 10.4 discusses this issue in more detail and shows that the enriched type system is necessary to achieve an effective sequential-to-concurrent code reuse.

6.5 Valid targets

Let's now turn to the problem of enforcing the atomicity of feature calls. The feature application semantics described in section 6.1.2 implies that the processor executing a routine *r* enjoys an exclusive access to the handlers of *r*'s formal arguments. To ensure this property, we have to prevent other clients from using these handlers in the meantime. Starting from SCOOP_97's solution of the problem, we define a set of criteria for valid targets and propose a generalised call validity rule which applies both to separate and non-separate calls. The new rule also eliminates the problem of void calls.

The *separate call rule* 4.2.2 of SCOOP_97 requires the target of a separate call to appear as formal argument of the enclosing routine. The rule is more restrictive than necessary: it prohibits separate calls on targets that do not appear as formal arguments but are handled by a processor that is locked in the context of the routine. Figure 6.6 illustrates this scenario. Only the call $x.f$ is valid because its target is a formal argument of *r*. All the other calls are invalid: $x.twin.f$ and $y.f$ are rejected by the compiler because their targets do not appear as formal arguments of *r*. But these calls are perfectly safe because $x.twin$ and *y* are not traitors: feature *twin* returns a clone of its target, handled by the same processor as *x*; that processor is already locked by *r*.

```
-- in class C
my_x, my_y: separate X

r (x: separate X)
    local
        y: separate X
    do
        x.f             -- Valid
        x.twin.f        -- Invalid although safe
        y := x.twin
        y.f             -- Invalid although safe
    end

s (x: separate X)
    do
        my_y.f          -- Invalid
        ...
    end
...
r (my_x)
s (my_x)
```

Figure 6.6: Limitations of the separate call rule in SCOOP_97

The separate call rule is clearly too restrictive. This is because it relies on a *syntactic* distinction between formal arguments and other entities, rather than capturing the essential *semantic* requirement: feature calls should only be permitted on targets whose handlers are locked in the

current context. This condition is trivially satisfied by all non-separate calls because their targets are handled by the current processor. The case of separate calls is a bit more complicated; there are several possibilities here. If the target of a call appears as formal argument of the enclosing routine, then the call is valid (this case is covered by the original rule). If the target is not a formal argument, the call may still be valid if one can statically demonstrate that the target is handled by the same processor as one of the formal arguments. The enriched type system comes in handy here: unqualified or qualified processor tags may be used to express the relative non-separateness of an entity with respect to a formal argument.

Let's relax the call validity rule to allow more flexibility in the treatment of feature calls without losing the atomicity guarantees. The following informal definition captures the refined requirements on targets.

Definition 6.5.1 (Valid target) *An expression exp may be used as target of a feature call in the context of routine r if and only if exp is attached and satisfies at least one of the following conditions:*

1. *exp is non-separate.*
2. *exp appears as formal argument of r.*
3. *exp has a qualified processor tag farg .**handler**, and farg is an attached formal argument of r.*
4. *exp has an unqualified processor tag p, and some attached formal argument of r has processor tag p.*

Conditions 1 and 2 mirror the two possibilities already present in SCOOP_97. Conditions 3 and 4 are new; these two cases increase the expressiveness of the model by accepting more separate calls. Note the requirement put on *farg*: it has to be attached. This permits to avoid SCOOP_97's inconsistency related to the locking of void arguments (see section 5.8.3); another, more fundamental reason behind it — the refined semantics of detachable and attached types — is discussed in section 7.1. Thanks to the *implicit type rule* 6.2.3, conditions 2 and 3 may be merged into one because the former implies the latter.

Figure 6.7 illustrates the applications of the refined rule. The call $y.f$ in the body of *r* is now correct because *y* is declared with the processor tag $x.\mathbf{handler}$ so that it is statically known to be handled by the same processor as *x*. Similarly, we know that *x.twin* is handled by *x*'s processor (we can show it formally using the rule 6.2.3 and the type combinator '$\star$') so the call *x.twin.f* is valid. The formal argument *x* of *s* has an unqualified processor tag *px*. Therefore, all calls to *s* expect an actual argument with the same processor tag *px*. The attribute *my_x* satisfies this requirement, therefore the call *s* (*my_x*) is correct. Within the body of *s*, any entity with the processor tag *px* may be safely used as target of a call because *s* locks the corresponding processor. Therefore, *my_y.f* is valid although *my_y* is not a formal argument.

Definition 6.5.2 formalises the notion of a *controlled expression*, based on the informal rules discussed above.

Definition 6.5.2 (Controlled expression) *An expression exp of type $T_{exp} = (\gamma, \alpha, C)$ is controlled if and only if exp is attached and satisfies one of the following conditions:*

```
-- in class C
px: PROCESSOR      -- unqualified processor tag
my_x, my_y: separate <px> X

r (x: separate X)
    local
        y: separate <x.handler> X   -- y is handled by x's processor
    do
        x.f                 -- Valid
        x.twin.f            -- Valid
        y := x.twin
        y.f                 -- Valid
    end

s (x: separate <px> X)
    do
        my_y.f              -- Valid
        ...
    end
...
r (my_x)
s (my_x)
```

Figure 6.7: Application of the refined call validity rule

1. *exp is non-separate, i.e.* $\alpha = \bullet$.

2. *exp appears in a routine r that has an attached formal argument farg and* α = *farg* .***handler***.

Rule 6.5.3 replaces the separate call rule (4.2.2) of SCOOP_97 and the validity rule 8.23.9 /VUNO/ of standard Eiffel ([53], p. 119), thus unifying the treatment of separate and non-separate calls, and prohibiting void targets. (A full eradication of void calls requires additional rules ensuring that all entities of attached types are properly initialised befor being used. This topic is not covered here.)

Definition 6.5.3 (Call validity rule) *Call exp*.*f* $(\bar{a})$ *appearing in class C is valid if and only if the following conditions hold:*

1. *exp is controlled.*

2. *exp's base class has a feature f exported to C, and the actual arguments* $\bar{a}$ *conform in number and type to the formal arguments of f.*

The rule is translated into the formal rules **T-CCallUnqual**, **T-QACallUnqual**, **T-QFCallUnqual**, **T-CCallQual**, **T-QACallQual**, and **T-QFCallQual**. Now it becomes clear why the rules for unqualified calls are stripped-down versions of **T-CCallQual**, **T-QACallQual**,

and **T-QFCallQual**: the controllability requirement is not listed among their premises because the implicit target of unqualified calls, **Current**, is trivially controlled in any context. (Furthermore, no type combinators are necessary because the client and the supplier are the same, therefore the actual and the expected types are evaluated in the same context.)

The new rule brings more flexibility by allowing more separate calls while preserving mutual exclusion and atomicity. The rule is not complete in that some safe calls are rejected. To deal with such cases, the object test mechanism described in section 6.7 can be used.

6.6 Object creation

A creation call on an entity results in creating a new object, placing it on a fresh processor — unless the entity's processor already exists — and attaching the entity to the object. No fresh processor needs to be created if the entity is non-separate, i.e. handled by the current processor (which obviously exists already), or if the entity has an explicit processor tag p and the corresponding processor has already been created (usually through an earlier creation instruction on another entity handled by p, or by an assignment to that entity). Figure 6.8 illustrates the different possibilities. Fresh processors only need to be created for x and y; z and my_x are placed

```
py: PROCESSOR -- declaration of processor tag py
x: separate X
my_x: X
y, z: separate <py> Y
 ...
create  x          -- Fresh processor is created for x.
create  my_x       -- my_x is placed on the current processor.
create  y.make     -- Fresh processor py is created for y.
create  z.make     -- Processor py already exists. z is placed on py.
```

Figure 6.8: Object creation

on existing processors. Note that y and z are placed on the same processor, in accordance with their declared type.

Type annotations enable the choice of a processor where the newly created object will be placed. This brings additional flexibility: without the enriched types, a factory object would be needed on each processor; instead of creating new separate objects, one would ask a factory to create them. The latter solution is much heavier and requires the knowledge of an appropriate factory in any context where an object is created. Our approach supports the object creation mechanism in its simple and natural form. However, the semantics of object creation is different than in sequential Eiffel because the call to a creation procedure may be separate; it must not interfere with calls on the target processor that may be executed in the meantime on behalf of another client. Therefore, an object creation requires the exclusive access to the target processor; it follows the semantics captured by rule 6.6.1.

Definition 6.6.1 (Object creation semantics)
A creation call ***create*** $x.cp$ $(...)$ *on the target* x *of type* (γ, α, X) *results in the following sequence of actions performed by the client's handler* P_c*:*

1. *Processor creation:*
   - *If x is separate, i.e.* $\alpha = \top$*, create a fresh processor* P_x*.*
   - *If x has explicit processor tag p, i.e.* $\alpha = p$*, then*
     - *if the corresponding processor* P_p *has already been created, take* $P_x = P_p$*.*
     - *if the corresponding processor has not been created yet, create a fresh processor* P_x *and set p to* P_x*.*
   - *If x is non-separate, i.e.* $\alpha = \bullet$*, take* $P_x = P_c$*.*
2. *Reservation: lock* P_x*.*
3. *Object creation: ask* P_x *to create a fresh instance of X using the creation procedure cp; attach x to the newly created object.*
4. *Release: release the lock on* P_x *acquired in the reservation step.*

Since the target processor P_x is locked for the exclusive use of the client during the creation of a fresh object and the execution of the creation procedure, there is no risk of interference by other clients. Step 3 needs some clarification: *x* is attached to the freshly created object as soon as the object is created but without waiting for the creation procedure to execute and terminate. As a result, the creation call is asynchronous (unless the target is non-separate); *x* points to a potentially inconsistent object before the creation procedure terminates but this is not harmful because the object cannot be accessed before its handler is released. This, in turn, cannot happen before *cp* terminates; the object is consistent at that point.

6.7 Handling false traitors

The enriched type system permits precise reasoning about the locality of objects. Nevertheless, static typing has its limitations: the exact run-time type of the object represented by an entity is often unknown at compile time. Modern programming languages deal with the incompleteness of their type systems by providing a run-time facility for type casting. Eiffel uses the *object test* mechanism for that purpose. An object test provides scoped binding of a fresh non-writable entity to the source expression; if the dynamic type of the source conforms to the declared type of the target entity, the test evaluates to **True**; it evaluates to **False** otherwise (the downcast fails). Figure 6.9 illustrates the use of an object test. Through the assignment *x* := *z*, the entity *x* becomes attached to an object of type *Z*. Nevertheless, the subsequent assignment *y* := *x* is rejected by the compiler even though the dynamic type of *x* conforms to the type of *y*. An object test helps to solve the problem: by downcasting *x* to *aux_y* (of type *Y*), we can use the object attached to *aux_y* as source of assignment to *y*. The use of *aux_y* is only valid in the **then** part of the conditional; *aux_y* is undefined in the **else** part. Note that *aux_y* is treated like a formal argument: it is only visible in the scope of the object test, and it is not writable, i.e. it cannot be used as target of assignments or creation instructions. An object test may also be used to perform assignments from detachable to attached entities. It is illegal to directly assign detachable *dz* to attached *z*; an object test using an auxiliary variable *aux_z* solves this problem. Both aspects of casting (non-voidness and conformance of class types) can be combined in a single object test, e.g. the type $(?, \bullet, X)$ can be cast to $(!, \bullet, Z)$.

```
-- in class C
x: X
y: Y                    -- Y conforms to X
z: Z                    -- Z conforms to Y and X
dz: ?Z
 ...
create z
x := z                  -- Valid assignment
y := x                  -- Invalid assignment

if {aux_y: Y} x then
    y := aux_y          -- Valid assignment
else
     ...                -- aux_y cannot be used here
end

z := dz                 -- Invalid assignment

if (aux_z: Z) dz then
    z := aux_z          -- Valid assignment
end
```

Figure 6.9: Object test

When a separate entity becomes attached to a non-separate object, the information about the object locality is lost: even though we know that the entity denotes a non-separate object at run time, we cannot assume it at compile time. Figure 6.10 and the corresponding code excerpt in figure 6.11 illustrate a typical scenario. Objects *o1* and *o2* are handled by the same processor P_1; object *o3* is handled by processor P_2. *o2* is known to *o3* as *y*; *o3* is known to *o1* as *my_x*, or *x* in the body of routine *r* when *my_x* is passed as actual argument. Since *x* is a separate entity, the result of $x.y$ is separate because $(!, \top, X) \star (!, \top, Y) = (!, \top, Y)$, even though the object denoted by $x.y$ is *o2* which is non-separate from *o1* (its dynamic type is $(!, \bullet, Y)$). An assignment from separate to non-separate, e.g. *my_y* := $x.y$, is invalid. If we allowed this assignment, *my_y* would become a traitor. In this particular scenario, it would be a *false traitor* because the object attached to it is non-separate. This information, however, is not available at compile time; therefore, we must prohibit such assignments to ensure type soundness.

To "detraitorise" a false traitor, we need to use the type information available at run time. We refine the semantics of object test to take into account the locality of its source. An object test succeeds if the run-time type of its source conforms in detachability, locality, and class type to the type of its target; it fails otherwise. This allows downcasting a separate entity to a non-separate one, provided that the entity represents a non-separate object at run time. Downcasts between types with different processor tags are possible, e.g. '$\top$' to '$\bullet$', '$\top$' to 'p', '$\bullet$' to 'x.**handler**', etc. Feature *s* in figure 6.11 is a corrected version of *r*; it uses an object test to perform the necessary downcast. Since $x.y$ has a dynamic type $(!, \bullet, Y)$, the object test is successful; *aux_y* is attached to *o2* and the assignment *my_y* := *aux_y* is valid. As a result, *my_y* is not a traitor, even though it holds a reference obtained initially through a separate query call.

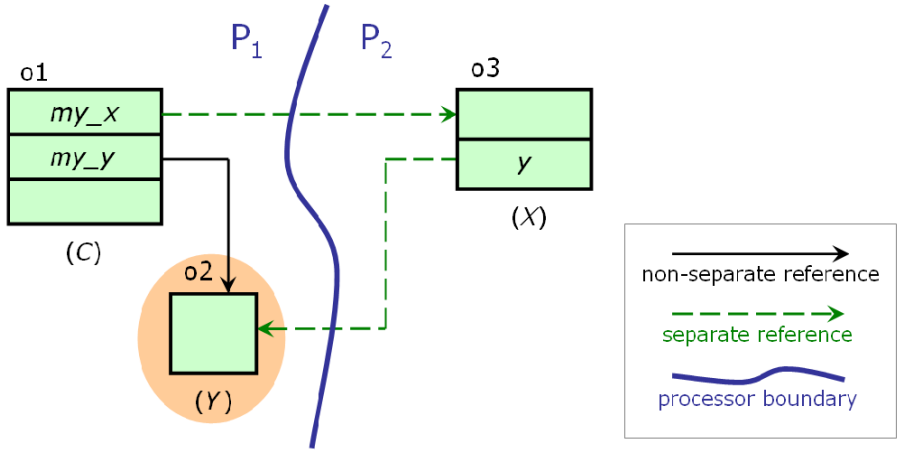


Figure 6.10: False traitor

Discussion

Meyer [96] suggests a locking semantics for object tests: a successful object test on a separate entity *x* blocks until the processor that handles *x* is acquired by the current processor. This semantic overloading supports the execution of separate calls without the need for an enclosing routine. (In fact, this can be achieved in a simpler way, described in section 9.3.4.) Unfortunately, besides compromising the primary role of the object test as a run-time conformance check, the blocking semantics severely limits the use of detachable and attached types: it is impossible to downcast a detachable separate entity to an attached type without performing any locking. Certain common scenarios, e.g. storing a detachable separate formal argument in an attribute for a later use, cannot be implemented. Additionally, object tests can only be performed on one entity at a time; atomic locking of several objects using a single object test is not supported.

The object test has replaced the *assignment attempt* mechanism which is still supported by existing Eiffel compilers. Essentially, an assignment attempt '?=' is like a standard assignment ':=' if the run-time type of its source conforms to the static type of its target; otherwise it assigns **Void** to the target. Since no compiler supports the object test mechanism yet, our implementation relies on the assignment attempt instead (see section 11.2).

6.8 Object import

The *deep_import* operation used in SCOOP_97 is a safe way to obtain a non-separate copy of an object structure. It yields a deep copy of the whole object structure, i.e. it copies recursively all substructures by following references and expanded attributes. As pointed out in section 4.2.5, using a shallow copy would introduce potential traitors.

But is it really necessary to copy the whole structure? Traitors may be introduced only if a non-separate reference is not followed. Separate references are not dangerous: the corresponding objects need not be copied because the type combination rules (see section 6.4) make sure that such objects are seen as separate. Therefore, a lightweight operation *import*, which only follows non-separate references and expanded attributes, is sufficient. (Applied to a non-

```
-- in class X
y: separate Y

-- in class C
my_x: separate X
my_y: Y

r (x: separate X)
    do
        my_y := x.y              -- Invalid assignment
        my_y.f
    end

s (x: separate X)
    do
        if {aux_y: Y} x.y then
            my_y := aux_y        -- Valid assignment
            my_y.f
        end
    end
...
r (my_x)
s (my_x)
```

Figure 6.11: Handling false traitors

separate object, *import* yields the same result as *twin*, i.e. it does not follow any references.) Figure 6.12 illustrates the effect of *import*. Assuming that the client *o1* executes the assignment $y := x.import$[1], where x is separate and y is not, all objects reachable from x via non-separate references, i.e. *o2* itself, *o3*, and *o4*, are copied onto processor P_1; y now references the object structure starting at *o2*'. Object *o5* is not copied because it is only reachable through a separate reference; this reference cannot become a traitor.

The import operation is a very convenient tool for "unseparating" an object structure. Nevertheless, in certain situations the imported structure does not sufficiently mirror the original structure. For example, field x of *o5* references object *o2*; as a result, $x.a.x = x.a$ holds but $y.a.x = y.a$ does not, although y is a copy of x. If the client wants to achieve a deep equality relation between the structures, it needs a different operation. But there are at least three possible semantics for such an operation:

- The relative separateness of objects is preserved; copies are placed on the same processors as their originals, e.g. a copy of *o5* would be placed on P_3. This operation is called *deep_import*.
- The whole object structure is placed on the client's processor; we may call it *flat_import*. In our example, *flat_import* would place the copy of *o5* on P_1. (Note that the

[1]We follow the style recommended in the Eiffel standard [53] and turn *import* into a query (like *twin*); the feature name should be understood as a noun.

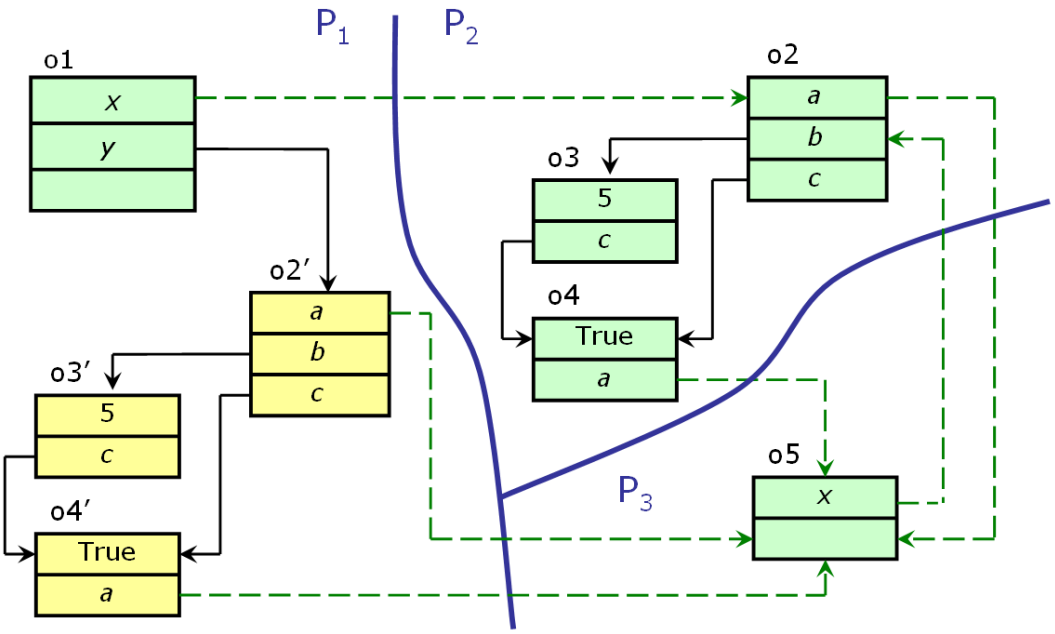


Figure 6.12: Importing an object structure

deep_import operation of SCOOP_97 has the same semantics.)

- The relative separateness of objects is preserved but copies are placed on fresh processors. In our example, a new processor P_4 would be created for the copy of *o5*. We may call this operation *independent_import* because the original object structure does not share processors with the copy. This property is interesting for independent parallel computations involving both structures.

We assume that *deep_import* will be used most often: it is the least expensive of the three, and it is closely related to *deep_twin*. The slightly heavier *flat_import* may prove useful in a distributed context where clients are interested in working on a local copy of data. The heaviest operation *independent_import* makes it possible to obtain structures that may be used concurrently without the risk of interlocking. These three features, together with the shallow *import*, seem to provide sufficient support for manipulating object structures in a concurrent context; however, we have not gathered enough practical experience to formulate a more precise claim.

6.9 Object equality

Assume that x is the root object of the original structure and y is its copy. If the copy has been obtained through the *twin* operation, *equal* (x, y) is true; if *deep_twin* has been used then *deep_equal* (x, y) holds. Let's see how the four import operations introduced in the previous section influence the equality relation between object structures, and how they relate to *twin* and *deep_twin* operations. One possibility is to ignore the relative separateness of objects and use only the reference equality, i.e. to preserve the standard "sequential" notion of equality [53]. The advantage of this solution is its simplicity. It seems, however, that the relative separateness of objects should be taken into account. Intuitively, a structure obtained through *deep_import* is not the same as a structure obtained through *flat_import*: identical operations applied to

each structure will have potentially different outcomes. Following this line of thought, i.e. accounting for the relative locality of objects, one may conclude the following: if y has been obtained through an *import* operation, then *equal* (x, y) might not hold; in fact, it will only hold if x is attached to a non-separate object (in which case the import is just like *twin*) or if x has no references to non-separate objects (in which case it is the only imported object). On the other hand, the *deep_import* operation preserves the relative locality of objects, so it will satisfy *deep_equal* (x, y). In general, neither *flat_import* nor *independent_import* satisfy *deep_equal* (x, y).

This enriched notion of equality between object structures is more complex than the standard one. Nevertheless, the close relation between *equal*, *twin*, and *import* on one hand, and *deep_equal*, *deep_twin*, and *deep_import* on the other hand suggests that the proposed approach is appropriate. This remains to be demonstrated in practice.

6.10 Expanded types

Expanded types may be used for expressing ownership relations between objects (e.g. a car engine *belongs* to a given car), and for emulating unique references. An expanded entity, that is an entity whose type is based on a class declared as

```
expanded class E
    ...
end
```

represents an object rather than a reference to an object. Expanded classes have copy semantics, i.e. their instances are always passed by copy. As a result, it is impossible to obtain multiple references to the same expanded object.

SCOOP_97 only accepts *fully expanded* objects as arguments or results of separate calls; a fully expanded object must not carry any non-separate references; see the consistency rule SC4 (4.2.9). Indeed, such references would become potential traitors if a copy of the object was placed on a different processor. Since most expanded classes (with a notable exception of *INTEGER*, *REAL*, *DOUBLE*, *BOOLEAN*, and *CHARACTER*) carry non-separate references, they become unusable in a concurrent context. We need to relax the consistency rules to accommodate arbitrary expanded types while preserving the safety guarantees.

Consider the three components of an expanded type (γ, α, E): the detachability tag, the processor tag, and the class type. E has to be an expanded class, otherwise the required copy semantics will not apply. An expanded object may be viewed as "sitting" inside the object that declares it; therefore, it must be handled by the same processor, i.e. $\alpha = \bullet$. Can an expanded entity be detachable? No, because it would then accept **Void** as possible value; hence $\gamma = !$. Rule 6.10.1 captures these requirements.

Definition 6.10.1 (Expanded type rule) *A type T_E based on an expanded class E is valid iff it is attached and non-separate, i.e. $T_E = (!, \bullet, E)$. T_E is then called an* expanded type.

Syntactically, the rule prohibits the use of separate annotations and the detachable tag '?' with expanded class types; no "detachable expanded" or "separate expanded" entities may be declared. But what happens if the result of a separate query call is expanded, or if a client passes

an expanded actual argument to a separate feature call? Can certain expressions involving separate calls be seen as separate and expanded at the same time? Consider figure 6.13. Expression $x.b$ is expanded because feature b in class X is of type $(!, \bullet, BOOLEAN)$; at the same time, it should also be separate because it is a result of a separate call: a naive calculation of its type using the type combinator '$\star$' yields $(!, \top, BOOLEAN)$. But the **if** ... **then** construct needs a non-separate boolean; furthermore, the rule 6.10.1 requires all expanded types to be non-separate. We need some way to "unseparate" such expressions. The simplest way is to apply implicitly the *import* operation described in section 6.8. Since *import* is applied implicitly, the expressions $x.b$ and $x.e$ are seen as non-separate by the client; the statement **if** $x.b$ **then** e := $x.e$ **end** is now correctly typed, and the assignment to e is valid. Similarly, the call $x.f\ (i)$ is valid because the expanded actual argument i is seen by the supplier x as non-separate: x receives a non-separate copy of i.

```
-- in class C
r (x: separate X; i: INTEGER)
    local
        e: E                -- E is (non-fully) expanded
    do
        if x.b then         -- x.b is non-separate
            e := x.e        -- x.e is non-separate
        end
        x.f (i)             -- i is seen as non-separate by x
    end


-- in class X
b: BOOLEAN
e: E
f (i: INTEGER)
    do
        ...
    end
```

Figure 6.13: Use of expanded types

The use of *import* prevents the creation of potential traitors when manipulating instances of arbitrary expanded classes; both fully-expanded and non-fully-expanded classes are treated correctly. Thus, the safety guarantees of SCOOP are preserved without the need to restrict the use of expanded types; the rule SC4 (4.2.9) is not necessary.

The type system must reflect the special status of expanded types. The rule **S-Subclass** is refined following the Eiffel conformance rule for expanded classes: an expanded type has no subtypes other than itself (see figure 6.20). It is also necessary to refine the type combinators '$\star$' and '$\otimes$' so that expanded types be invariant under both operators, i.e. for an arbitrary type T and an expanded class type E, $T \star (!, \bullet, E) = (!, \bullet, E)$ and $T \otimes (!, \bullet, E) = (!, \bullet, E)$; see figure 6.14. The refined rules use an auxiliary function $\mathbf{isExpanded} : Class(\Gamma) \longrightarrow \mathbf{Bool}$ which tells whether a class is expanded.

(Result type) $\star : Type \times Type \longrightarrow Type$

$$(\gamma, \alpha, C) \star (\lambda, \beta, D) = \begin{cases} (\lambda, \beta, D) & \text{if } isExpanded(D) \\ (\lambda, \alpha, D) & \text{if } \neg isExpanded(D) \ \wedge \ \beta \in \{\bullet, \mathbf{Current.handler}\} \\ (\lambda, \top, D) & \text{otherwise} \end{cases} \quad \text{(TC-}\star\text{)}$$

(Argument type) $\otimes : Type \times Type \longrightarrow Type$

$$(\gamma, \alpha, C) \otimes (\lambda, \beta, D) = \begin{cases} (\lambda, \beta, D) & \text{if } isExpanded(D) \\ (\lambda, \alpha, D) & \text{if } \neg isExpanded(D) \ \wedge \ \alpha \neq \top \\ & \quad \wedge \beta \in \{\bullet, \mathbf{Current.handler}\} \\ (\lambda, \top, D) & \text{if } \neg isExpanded(D) \ \wedge \ \beta = \top \\ (\lambda, \bot, D) & \text{otherwise} \end{cases} \quad \text{(TC-}\otimes\text{)}$$

Figure 6.14: Refined type combinators

6.11 Formalisation of the type system

In this section, we formalise the compile-time aspects of SCOOP's type system. We define the syntax of the programming language, typing environments, well-formedness conditions, subtyping, and the type rules for statements, expressions, features, classes, and programs.

6.11.1 SCOOP$_C$ programs

$SCOOP_C$ (where *C* stands for "core") is a subset of SCOOP that includes the following constructs: classes, expanded classes, features, contracts, single inheritance, polymorphism, feature redefinition (with invariant typing of arguments and result), attached and detachable types, object creation, feature calls, assignments, conditionals, object tests, and loops. Multiple inheritance, genericity, deferred classes, covariant and contravariant typing of redefined features, agents, and once features are excluded for the sake of simplicity; chapter 9 demonstrates these mechanisms are dealt with.

Figure 6.15 summarises the syntax. We follow the convention that $SCOOP_C$ keywords appear as **keyword**, identifiers as $identifier$, and non-terminals as $NonTerminal$. Identifiers with the suffix id, e.g. $classid$, indicate newly defined constructs; identifiers with the suffix $name$, e.g. $classname$, indicate previously declared constructs.

A program (sort **Program**) contains class declarations and a root creation instruction composed of a root class name and a creation procedure in a call-like dot notation.

A class declaration (**ClassDecl**) contains the name of the class, the name of its direct ancestor (it may be the predefined class *ANY*), names of creation procedures and unqualified processor tags, feature declarations (attributes and routines), and an invariant.

A routine declaration (**RoutineDecl**) is composed of a signature (name, formal argument declarations, result type), a precondition, local variable declarations, a body (composed of statements), and a postcondition. We assume that all assertions are fully specified, i.e. every routine

lists a precondition and a postcondition, and every class has an explicit invariant; they may be trivially **True**.

An attribute declaration consists of a name and a type; so do local variable and formal argument declarations.

A type (**Type**) consists of a detachable tag, a processor tag, and a class type.

The set of expressions (**Expr**) includes the literals **Current**, **Void**, **True**, **False**, and **Result**, as well as integers, local variables, formal arguments, unqualified and qualified query calls, object test, and expressions built using the predefined operators '**not**', '**and**', '**or**', '**implies**', and '='.

The set of statements (**Stmt**) includes assignments, procedure calls, object creation, conditionals, and loops.

We assume that all identifiers in a program are unique, except for identifiers and formal parameters of redefined features, and local variables of different features. The disambiguation may be achieved by preceding each identifier with the name of the class or the feature where it is declared. For simplicity, we omit the prefixes in the examples.

6.11.2 Typing environments

Type checking of a program p is performed in a type environment Γ which contains the class hierarchy derived from p, enriched with ANY and $NONE$, and the type definitions of all features; the local variables and the formal arguments of the enclosing routine to their types. To formalise typing environments, we follow the approach proposed by Drossopoulou and Eisenbach [52].

The formal syntax for environments is given in figure 6.16. $StandardEnv$ includes the predefined classes *ANY* and *NONE* which list no ancestors and have no features. A class declaration introduces a new class as a subclass of another class. Feature types appear in class declarations. Assertions and feature bodies are excluded; they are found in the program rather than the environment. Local variable declarations introduce variables of a given type; **Result** is treated as an implicit local variable of every function. Formal argument declarations introduce formal arguments of a given type. **Current** is treated as implicit formal argument; therefore, it is always included in the typing environment (in class C, the type of **Current** is $(!, \bullet, C)$). Fresh variables introduced by object tests are treated as formal arguments and carried in the typing environment (see rule **T-ObjectTest** in figure 6.23).

Subclassing

The inference rules in figure 6.17 define the conformance relation $\sqsubseteq$ on class types in an environment Γ. The conformance relation is often referred to as *subclassing*. An assertion

$$\Gamma = \Gamma',\ \textbf{class}\ C\ \textbf{inherit}\ C' \ldots\ \textbf{end}, \Gamma''$$

should be understood as "Γ contains a declaration of class C as a subclass of C'". Every class in Γ is its own subclass, as indicated by the assertion $\Gamma \vdash C \sqsubseteq C$ in rule **C-Class**. Every class is also a subclass of its immediate superclass (rule **C-Subclass**). Subclassing is transitive (rule **C-Trans**); as a result, all class types conform to *ANY*. Conversely, *NONE* conforms to every class type (rule **C-SubNone**).

$$
\begin{array}{rcl}
Program & ::= & ClassDecl^{*}\ \ RootCreation \\
RootCreation & ::= & classname\ \textbf{.}\ routname \\
ClassDecl & ::= & [\textbf{expanded}]\ \textbf{class}\ classid\ \textbf{inherit}\ classname\ \textbf{create}\ routname^{*} \\
 & & \textbf{feature}\ tagid^{*}\ FeatureDecl^{*}\ \textbf{invariant}\ Expr\ \textbf{end} \\
FeatureDecl & ::= & AttributeDecl \mid RoutineDecl \\
AttributeDecl & ::= & VarDecl \\
RoutineDecl & ::= & routid\ [(\ ArgDecl^{+}\textbf{)}]\ [\textbf{:}\ Type] \\
 & & \textbf{require}\ Expr\ [\textbf{local}\ VarDecl^{+}]\ \textbf{do}\ Stmts\ \textbf{ensure}\ Expr\ \textbf{end} \\
ArgDecl & ::= & argid\ \textbf{:}\ Type \\
VarDecl & ::= & varid\ \textbf{:}\ Type \\
Stmts & ::= & \epsilon \mid Stmts\ \textbf{;}\ Stmt \\
Stmt & ::= & Var\ \textbf{:=}\ Expr \mid Expr\textbf{.}routname\ [(\ Expr^{+}\textbf{)}] \\
 & \mid & \textbf{create}\ Var\textbf{.}routname\ [(\ Expr^{+}\textbf{)}] \\
 & \mid & \textbf{if}\ Expr\ \textbf{then}\ Stmts\ \textbf{else}\ Stmts\ \textbf{end} \\
 & \mid & \textbf{from}\ Stmts\ \textbf{until}\ Expr\ \textbf{loop}\ Stmts\ \textbf{end} \\
Expr & ::= & Value \mid Var \mid Expr\textbf{.}attrname \mid Expr\textbf{.}routname\ [(\ Expr^{+}\textbf{)}] \\
 & \mid & \textbf{not}\ Expr \mid Expr\ \textbf{and}\ Expr \mid Expr\ \textbf{or}\ Expr \\
 & \mid & Expr\ \textbf{implies}\ Expr \mid Expr\ \textbf{=}\ Expr \mid \{ArgDecl\}\ Expr \\
Var & ::= & varname \mid \textbf{Result} \\
Value & ::= & argname \mid \textbf{Current} \mid \textbf{Void} \mid \textbf{True} \mid \textbf{False} \\
 & \mid & i\ \ (i \in \mathbb{Z}) \mid c\ \ (c \in Character) \\
ProcTag & ::= & \bullet \mid \top \mid tagname \mid argname\textbf{.handler} \\
DetachTag & ::= & ! \mid ? \\
Type & ::= & \textbf{(}\ DetachTag, ProcTag, classname\ \textbf{)}
\end{array}
$$

Figure 6.15: $SCOOP_C$ programs

$$
\begin{array}{rll}
Env & ::= & StandardEnv \mid Env\ ;\ Decl \\
StandardEnv & ::= & \textbf{class ANY end},\ \textbf{class NONE end} \\
Decl & ::= & \textbf{class}\ classid\ \textbf{inherit}\ classname\ \textbf{create}\ routname^* \\
 & & \textbf{feature}\ tagid^*\ \ (attrid : Type)^*\ \ (routid : RoutType)^* \\
 & & \textbf{end} \\
 & & \mid varid : Type \\
 & & \mid argid : Type \\
ClassType & ::= & classname \mid \textbf{ANY} \\
DetachTag & ::= & ! \mid ? \\
ProcTag & ::= & \bullet \mid \top \mid tagname \mid argname.\textbf{handler} \\
Type & ::= & (DetachTag, ProcTag, ClassType) \\
RoutType & ::= & ArgType \longrightarrow Type \mid ArgType \longrightarrow \varnothing \\
ArgType & ::= & Type\ (\times\ Type)^* \mid \varnothing
\end{array}
$$

Figure 6.16: $SCOOP_C$ environments

6.11.3 Valid types

Figure 6.18 gives the well-formedness rules for types. Class types and processor tags are required in type declarations; routine types are required in routine declarations. The assertion $\Gamma \vdash C \diamond_{ClassType}$ means that C is a valid class type. The assertion $\Gamma \vdash pt \diamond_{ProcTag}$ means that pt is a valid processor tag. The assertion $\Gamma \vdash T \diamond_{Type}$ means that T is a valid type. The assertion $\Gamma \vdash RT \diamond_{RoutType}$ means that RT is a valid routine type. Figure 6.19 defines the auxiliary functions used in the well-formedness rules.

6.11.4 Subtyping

Section 6.3 discussed the conformance relation between SCOOP types. Recall that a type U is a subtype of T, expressed as $U \preceq T$, if and only if the three components of U — detachable tag, processor tag, and class type — conform to the corresponding components of T. The conformance of class types has been defined in the previous section; the conformance of processor tags is captured in figure 6.1. Detachable tag '!' conforms to '?'. The four rules in figure 6.20 capture the subtype relation in a formal manner. Rule **S-Subclass** says that a type may be viewed as based on a superclass of its class type, with all the other components unchanged; one may also deduce $T \preceq T$. The rule prohibits subtyping of an expanded type by another type, i.e. an expanded type is a unique subtype of itself. Rule **S-Top** enables the generalisation of the processor tag to '$\top$'; conversely, **S-Bottom** captures the conformance of '$\bot$' to any processor tag. Rule **S-Attached** formalises the conformance between attached and detachable types.

Note the absence of subsumption and transitivity rules; we avoid them here to make type-

$$\frac{\Gamma \ = \ \Gamma', \ \textbf{class}\ C\ \textbf{inherit}\ C' \ ...\ \textbf{end}, \Gamma''}{\Gamma \vdash C \sqsubseteq C} \quad \text{(C-Class)}$$

$$\frac{\Gamma \ = \ \Gamma', \ \textbf{class}\ C\ \textbf{inherit}\ C' \ ...\ \textbf{end}, \Gamma''}{\Gamma \vdash C \sqsubseteq C'} \quad \text{(C-Subclass)}$$

$$\frac{\Gamma \vdash C \sqsubseteq C' \qquad \Gamma \vdash C' \sqsubseteq C''}{\Gamma \vdash C \sqsubseteq C''} \quad \text{(C-Trans)}$$

$$\frac{}{\vdash ANY \sqsubseteq ANY} \quad \text{(C-Any)}$$

$$\frac{}{\vdash NONE \sqsubseteq C} \quad \text{(C-None)}$$

Figure 6.17: Class conformance

checking rules deterministic. We achieve the necessary reflexivity and transitivity by relying on subclassing which is itself reflexive and transitive.

It is easy to see that the most general type is $(?, \top, ANY)$; all other types conform to it. There is no type that conforms to all other types. Observe the compatibility of rules **S-Subclass** and **S-Attached** with the standard Eiffel rules [53].

6.11.5 Well-formed environments

The relations $\sqsubseteq$ and $\preceq$ are computable in any environment. Figure 6.21 describes the requirements for the well-formedness of variable and class declarations. The judgement $\Gamma \vdash \Gamma' \diamond$ says that all the declarations in the environment Γ' are well-formed given the environment Γ; the latter, potentially larger, is needed because of forward declarations, i.e. the use of types based on classes declared later in the program. To check $\Gamma \vdash \Gamma' \diamond$, relations $\sqsubseteq$ and $\preceq$ have to be established for Γ, and their acyclicity demonstrated; secondly, the well-formedness of each declaration in Γ' must be demonstrated. The ultimate goal is to establish $\Gamma \vdash \Gamma \diamond$ where Γ includes all class declarations. An empty environment is well-formed (rule **WF-EmptyEnv**). Rule **WF-VarArgEnv** states that a local variable or a formal argument must have a valid type and be declared at most once. A definition lookup function $\Gamma(id)$ which returns the definition of the identifier id in Γ is necessary for reasoning about the well-formedness of declarations. For an environment Γ with unique definitions for every identifier, we define $\Gamma(id)$ as follows:

- $\Gamma(x) = T$ if and only if $\Gamma = \Gamma', x : T, \Gamma''$

- $\Gamma(C) =$ **class** C **inherit** C' **create** $cr_1, ..., cr_k$ **feature** $pt_1, ..., pt_l$, $a_1 : T_1, ..., a_m : T_m,\ r_1 : RT_1, ...,\ r_n : RT_n$ **end**

  if and only if

$$\frac{\Gamma \vdash C \sqsubseteq C}{\Gamma \vdash C \ \diamond_{ClassType}} \quad \text{(WF-ClassType)}$$

$$\frac{}{\Gamma \vdash \bullet \ \diamond_{ProcTag}} \quad \text{(WF-BulletPTag)}$$

$$\frac{}{\Gamma \vdash \top \ \diamond_{ProcTag}} \quad \text{(WF-TopPTag)}$$

$$\frac{\Gamma = \Gamma', \textbf{class}\ C\ ...\ \textbf{feature}\ ..., pt, ...\ \textbf{end}, \Gamma''}{\Gamma \vdash pt \ \diamond_{ProcTag}} \quad \text{(WF-UnqualifiedPTag)}$$

$$\frac{}{\Gamma \vdash \textbf{Current.handler} \ \diamond_{ProcTag}} \quad \text{(WF-CurrentPTag)}$$

$$\frac{\Gamma = \Gamma', x : T, \Gamma'' \quad \Gamma \vdash \neg isWritable(x),\ \ \Gamma \vdash isAttached(T)}{\Gamma \vdash x.\textbf{handler} \ \diamond_{ProcTag}} \quad \text{(WF-QualifiedPTag)}$$

$$\frac{\gamma \in \{!, ?\},\ \ \Gamma \vdash \alpha \ \diamond_{ProcTag} \quad \Gamma \vdash C \ \diamond_{ClassType},\ \ \Gamma \vdash \neg isExpanded(C)}{\Gamma \vdash (\gamma, \alpha, C) \ \diamond_{Type}} \quad \text{(WF-ReferenceType)}$$

$$\frac{\Gamma \vdash C \ \diamond_{ClassType},\ \ \Gamma \vdash isExpanded(C)}{\Gamma \vdash (!, \bullet, C) \ \diamond_{Type}} \quad \text{(WF-ExpandedType)}$$

$$\frac{\Gamma \vdash T_i \ \diamond_{Type} \quad i \in \{1..n\}, n \geq 0}{\Gamma \vdash T_1 \times ... \times T_n \longrightarrow \varnothing \ \diamond_{RoutType}} \quad \text{(WF-ProcedureType)}$$

$$\frac{\Gamma \vdash T \ \diamond_{Type} \quad \Gamma \vdash T_i \ \diamond_{Type} \quad i \in \{1..n\}, n \geq 0}{\Gamma \vdash T_1 \times ... \times T_n \longrightarrow T \ \diamond_{RoutType}} \quad \text{(WF-FunctionType)}$$

Figure 6.18: Well-formedness of types

$$\frac{}{\Gamma \vdash isAttached((!, \alpha, C))} \quad \text{(A-isAttached)}$$

$$\Gamma \vdash ClassType(\gamma, \alpha, C) = C \quad \text{(A-ClassType)}$$

$$\Gamma \vdash isExpanded(C) \iff p = p', \textbf{expanded class } C \text{ ... } \textbf{end}, p'' \quad \text{(A-isExpanded)}$$

$$\Gamma \vdash isFormalArgument(x) \iff x = \textbf{Current} \vee (\Gamma = \Gamma', x : T, \Gamma'' \wedge p = p', ...\textbf{class } C \text{ ... } \textbf{feature } ..., f(..., x : T, ...)..., ... \textbf{ end}, p'') \quad \text{(A-isFormalArgument)}$$

$$\Gamma \vdash isWritable(x) \iff \Gamma = \Gamma', x : T, \Gamma'' \wedge \Gamma \vdash \neg isFormalArgument(x) \quad \text{(A-isWritable)}$$

$$\frac{\Gamma \vdash farg : (!, \alpha_{farg}, C_{farg}), \;\; \Gamma \vdash isFormalArgument(farg) \quad \Gamma \vdash C \diamond_{ClassType}}{\Gamma \vdash isControlled((!, farg.\textbf{handler}, C))} \quad \text{(A-isControlled)}$$

Figure 6.19: Auxiliary functions

$$\frac{\Gamma \vdash C \sqsubseteq C', \;\; \Gamma \vdash isExpanded(C') \Longrightarrow C = C'}{\Gamma \vdash (\gamma, \alpha, C) \preceq (\gamma, \alpha, C')} \quad \text{(S-Subclass)}$$

$$\frac{\Gamma \vdash C \sqsubseteq C'}{\Gamma \vdash (\gamma, \alpha, C) \preceq (\gamma, \top, C')} \quad \text{(S-Top)}$$

$$\frac{\Gamma \vdash C \sqsubseteq C'}{\Gamma \vdash (\gamma, \bot, C) \preceq (\gamma, \alpha, C')} \quad \text{(S-Bottom)}$$

$$\frac{\Gamma \vdash (?, \alpha, C) \preceq (?, \beta, C')}{\Gamma \vdash (!, \alpha, C) \preceq (?, \beta, C')} \quad \text{(S-Attached)}$$

Figure 6.20: Subtyping

$$\frac{}{\Gamma \vdash \epsilon \ \diamond} \qquad \text{(WF-EmptyEnv)}$$

$$\frac{\Gamma \vdash \Gamma' \ \diamond \quad \Gamma \vdash T \ \diamond_{Type} \quad x \notin dom(\Gamma')}{\Gamma \vdash \Gamma', x : T \ \diamond} \qquad \text{(WF-VarArgEnv)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \Gamma' \ \diamond, \ \ C \notin dom(\Gamma') \\ \Gamma \vdash C' \sqsubseteq C', \ \ \Gamma \vdash C' \not\sqsubseteq C \\ \Gamma \vdash \forall j \in \{1..m\}.\ T_j \diamond_{Type}, \ \ \Gamma \vdash \forall j \in \{1..n\}.\ RT_j \diamond_{RoutType} \\ a_i = a_j \Longrightarrow i = j \ \ i, j \in \{1..m\} \\ a_i = a_j \Longrightarrow i = j \ \ i, j \in \{1..n\} \\ \Gamma \vdash \forall j \in \{1..m\}.\ (FeatureType(C', a_j) = T_j \ \vee \ FeatureType(C', a_j) = undef) \\ \Gamma \vdash \forall j \in \{1..n\}.\ (FeatureType(C', r_j) = RT_j \ \vee \ FeatureType(C', r_j) = undef) \\ \Gamma \vdash \forall j \in \{1..l\}.\ cr_j \in Procedure(C)\end{array}}{\begin{array}{c}\Gamma \vdash \Gamma', \textbf{class}\ C\ \textbf{inherit}\ C'\ \textbf{create}\ cr_1, ..., cr_k\ \textbf{feature}\ pt_1, ..., pt_l, \\ a_1 : T_1, ..., a_m : T_m, \ r_1 : RT_1, ..., \ r_n : RT_n \ \ \textbf{end} \ \diamond\end{array}} \qquad \text{(WF-ClassEnv)}$$

Figure 6.21: Well-formed environments

$\Gamma = \Gamma',\ \textbf{class}\ C\ \textbf{inherit}\ C'\ \textbf{create}\ cr_1, ..., cr_k\ \textbf{feature}\ pt_1, ..., pt_l,$
$a_1 : T_1, ..., a_m : T_m,\ r_1 : RT_1, ...,\ r_n : RT_n\ \ \textbf{end}, \Gamma''$

- $\Gamma(id) = undef$ otherwise

Rule **WF-ClassEnv** says that a declaration of class C as subclass of C' is well-formed if and only if the following conditions are satisfied:

- C has not been previously declared.
- The declaration does not introduce a cycle in the subclassing relation.
- All types and routine types occurring in the declaration are valid.
- All feature names are unique. (Feature overloading is prohibited.)
- The types of all redefined features in C are identical with the types of the corresponding features in C'. (This enforces the invariance of argument and result types.)
- Each creation procedure is defined in C or one of its ancestors.

To check this, we need a few auxiliary functions. Let $Class(\Gamma)$ represent the set of all classes defined within the environment Γ. $C \in Class(\Gamma)$ if and only if $\Gamma \vdash C \sqsubseteq C$. For an environment Γ with a class declaration for C, i.e.

$\Gamma = \Gamma',\ \textbf{class}\ C\ \textbf{inherit}\ C'\ \textbf{create}\ cr_1, ..., cr_k\ \textbf{feature}\ a_1 : T_1, ..., a_l : T_l,\ r_1 : RT_1, ...,\ r_m : RT_m\ \ \textbf{end}, \Gamma''$

the feature lookup function $FeatureType : Class(\Gamma) \times Feature \longrightarrow Type \cup RoutType$ is defined as

- $FeatureType(ANY, f) = undef$ for any f
- $FeatureType(NONE, f) = undef$ for any f
- $FeatureType(C, f) = T_j$ iff $f = a_j$
- $FeatureType(C, f) = RT_j$ iff $f = r_j$
- $FeatureType(C, f) = FeatureType(C', f)$ otherwise

the set of procedure names $Procedure : Class(\Gamma) \longrightarrow Id$ as

- $Procedure(C) = \{r_1, ..., r_m\} \cup Procedure(C')$

and the set of creation procedure names $Creator : Class(\Gamma) \longrightarrow Id$ as

- $Creator(C) = \{cr_1, ..., cr_k\}$

The following properties of well-formed environments are easy to demonstrate:

- Subtyping and subclassing relations are reflexive, transitive, and antisymmetric.
- The class hierarchy forms a complete lattice with the top element *ANY* and the bottom element *NONE*.
- The type hierarchy does not form a lattice; there is a top element $(?, \top, ANY)$ but no bottom element.
- If two types are in the subtype relation, them their class types are in the subclass relation, i.e. $U \preceq T \Leftrightarrow ClassType(U) \sqsubseteq ClassType(T)$.

From now on, all environments are implicitly assumed to be well-formed.

6.11.6 Type rules

Figures 6.22 – 6.24 list the type rules for $SCOOP_C$. The rules with a conclusion of the form $\Gamma \vdash s \diamond$ should be understood as "s is well-typed in Γ"; rules with a conclusion $\Gamma \vdash e : T$ should be understood as "e is well-typed in Γ and has the type T".

Figure 6.22 describes the types for expressions: primitive values, variables, expressions built on predefined operators, and query calls. The predefined values **True** and **False** have the boolean type. Integer literals have the integer type, and character literals have the character type. **Void** has the type $(?, \bot, NONE)$. Local variables and formal arguments of the current feature (including **Current**) have the type declared in the environment (rule **T-Var**). Rule **T-Implicit** gives a non-writable attached entity a type based on an implicit qualified processor tag; it has been discussed in detail in section 6.2.2. Expressions based on the predefined operators '**not**', '**and**', '**or**', '**implies**', and '**=**' have the boolean type. Rules **T-QACallUnqual** and

T-QFCallUnqual say that an unqualified query call is well-typed if and only if the given query is defined for the current class, and the actual arguments of the call conform in number and type to the expected formals; the call expression has the same type as the result type of the query. Rules **T-QACallQual** and **T-QFCallQual** for qualified query calls require the target to be controlled (see section 6.5 and the auxiliary function $isControlled$ in figure 6.19), the query to be defined for the target's base class, and the actual arguments to conform in number and type to the expected formals; the types of the latter are combined with the target's type using the type combinator '$\otimes$'. The type of the call expression is calculated by combining the result type of the query with the target's type using the combinator '$\star$'.

Figure 6.23 describes the types for statements: sequences, conditionals, loops, assignments, command calls, creation instructions. A sequence of statements is well-typed if its components are (rule **T-Seq**). A conditional statement is well-typed if the condition has the boolean type and both **then** and **else** parts are well-typed (rule **T-If**). We also provide the rule **T-ObjectTest** for conditionals that use an object test; such conditionals are well-typed if the target of the object test is a fresh variable of a valid type, both the source of the object test and the **else** part of the conditional are well-typed, and the **then** part is well-typed in the environment enriched with the target[2]. A loop is well-typed if the exit condition has the boolean type, and both the **from** part and the body are well-typed (rule **T-Loop**). An assignment is well-typed if its target is writable, i.e. an attribute or a local variable (see the auxiliary function $isWritable$ in figure 6.19), and the type of the source conforms to the type of the target (rule **T-Assign**). Rules **T-CCallUnqual** and **T-QCallUnqual** are similar to the corresponding rules **T-QFCallUnqual** and **T-QFCallQual** for queries but do not involve result types. The rule for creation instructions **T-Create** requires the target to be writable, the creation procedure to be defined for the target's base class and marked as a creator, and the actual arguments to conform in number and type to the expected formals; the types of the latter are combined with the target's type using the type combinator '$\otimes$'.

The well-formedness rules for routines, class declarations, and programs are given in figure 6.24. Rule **T-Procedure** states that a procedure declaration is well-typed and if and only if the formal arguments and the local variables have valid types, its precondition and postcondition have the boolean type in the environment enriched with the formal arguments, and the body is well-typed in the environment enriched with the formal arguments and the local variables. The type of the procedure is the cartesian product of the formal arguments' types. Similarly, the rule **T-Function** states that a function declaration is well-typed and if and only if the formal arguments and the local variables have valid types, the result type is valid, the precondition has the boolean type in the environment enriched with the formal arguments, the postcondition has the boolean type in the environment enriched with the formal arguments and the local variable **Result**, and the body is well-typed in the environment enriched with the formal arguments, the local variables, and **Result**.

A class is well-formed if all its attributes have valid types, all its routine declarations are well-formed and have valid types, and the invariant has the boolean type in the environment enriched with the declaration of **Current**.

Finally, a program is well-formed and complete, i.e. $\Gamma \vdash p \diamond$, if it provides a class declaration for each class in $Class(\Gamma) \setminus \{ANY, NONE\}$, it contains at most one class declaration for

[2]Rules for other CAPs (Certified Attachment Patterns) involving object tests may be built following a similar schema; we omit them here for conciseness.

the same class name, all classes are well-formed, and the root creation clause consists of a valid root class name and a root creation procedure that is listed among the creators of the root class.

6.12 Properties of the type system

This section demonstrates the use of our type system. An example SCOOP$_C$ program is type-checked; we argue informally the soundness of the type system, in the sense that the execution of a correctly typed program will not introduce traitors. We also prove two important lemmas:

- *Monotonicity of separate references*
  An expression e of the form $f_0.f_1.\cdots.f_n,\ n \geq 0$, is separate if some f_i is separate and no f_j is expanded, for all $j \geq i$.

- *Correct wrapping of separate calls*
  If an expression e is separate, i.e. $\Gamma \vdash e : (\gamma, \alpha, C) \wedge \alpha \notin \{\bullet, \mathbf{Current.handler}\}$, then a feature call on e may occur only in the precondition, the postcondition, or the body of a routine r which takes an attached formal argument $farg$ such that e and is non-separate from $farg$.

The first lemma helps proving the absence of traitors; the second one helps proving that separate calls are only possible on locked targets. No formal proof of soundness is provided, however, due to the lack of an operational semantics for SCOOP$_C$.

Note that soundness means the absence of traitors but not the absence of void calls. As pointed out in section 6.5, our type rules are strong enough to ensure that entities of attached types remain non-void once they have been properly initialised; but we provide no formalisation of the initialisation rules. A full treatment of the topic is beyond the scope of this work. For simplicity, we may assume that a compiler performs additional checks, e.g. requires that every routine initialise its attached local variables before using them, and that every creation procedure initialise the attached attributes of its class before issuing any calls. Entities of expanded types can be viewed as self-initialising. (A similar solution is used in the Eiffel standard [53].)

6.12.1 Example

We use a client–buffer scenario where the client object creates a separate buffer, stores an element in the buffer, then retrieves an element. This simple program, including classes $CLIENT$, $BUFFER$, and $BOOLEAN$, demonstrates type-checking of several SCOOP constructs: creation of separate objects, feature calls on separate and non-separate targets, assignments, assertions, qualified processor tags. The corresponding SCOOP$_C$ program p is given in figures 6.25–6.26. Of particular interest is the qualified processor tag used in the routine *store*, which is necessary to demonstrate the correctness of the separate call *a_buffer*.*put*(b). Routine *put* in class *BUFFER* takes a non-separate formal argument, whereas the actual argument of the call is separate; the enriched type of the latter, together with an application of the type combinator '$\otimes$' permits concluding the relative non-separateness of the target and the actual argument. The type combinator '$\star$' is used to evaluate types of separate feature call results from the point of view of the client. Furthermore, the example involves separate calls in preconditions and postconditions.

$$\frac{}{\Gamma \vdash \textbf{True} : (!, \bullet, BOOLEAN)} \quad \text{(T-True)}$$

$$\frac{}{\Gamma \vdash \textbf{False} : (!, \bullet, BOOLEAN)} \quad \text{(T-False)}$$

$$\frac{}{\Gamma \vdash \textbf{Void} : (?, \bot, NONE)} \quad \text{(T-Void)}$$

$$\frac{i \in \mathbb{Z}}{\Gamma \vdash i : (!, \bullet, INTEGER)} \quad \text{(T-Int)}$$

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash\ x : \Gamma(x)} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash e : (!, \alpha, C),\ \ \Gamma \vdash \neg isWritable(e)}{\Gamma \vdash e : (!, e.\textbf{handler}, C)} \quad \text{(T-Implicit)}$$

$$\frac{\Gamma \vdash e_1 : T,\ \ \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \textbf{ = } e_2 : (!, \bullet, BOOLEAN)} \quad \text{(T-Equal)}$$

$$\frac{\Gamma \vdash e : (!, \bullet, BOOLEAN)}{\Gamma \vdash \textbf{not}\ e : (!, \bullet, BOOLEAN)} \quad \text{(T-Not)}$$

$$\frac{\Gamma \vdash e_1 : (!, \bullet, BOOLEAN),\ \ \Gamma \vdash e_2 : (!, \bullet, BOOLEAN)}{\Gamma \vdash e_1\ \textbf{and}\ e_2 : (!, \bullet, BOOLEAN)} \quad \text{(T-And)}$$

$$\frac{\Gamma \vdash e_1 : (!, \bullet, BOOLEAN),\ \ \Gamma \vdash e_2 : (!, \bullet, BOOLEAN)}{\Gamma \vdash e_1\ \textbf{or}\ e_2 : (!, \bullet, BOOLEAN)} \quad \text{(T-Or)}$$

$$\frac{\Gamma \vdash e_1 : (!, \bullet, BOOLEAN),\ \ \Gamma \vdash e_2 : (!, \bullet, BOOLEAN)}{\Gamma \vdash e_1\ \textbf{implies}\ e_2 : (!, \bullet, BOOLEAN)} \quad \text{(T-Implies)}$$

$$\frac{\Gamma \vdash \textbf{Current} : (!, \bullet, C),\ \ \Gamma \vdash FeatureType(C, f) = T}{\Gamma \vdash f : T} \quad \text{(T-QACallUnqual)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : T_e,\ \ \Gamma \vdash isControlled(T_e),\ \ \Gamma \vdash ClassType(T_e) = C \\ \Gamma \vdash FeatureType(C, f) = T\end{array}}{\Gamma \vdash e.f : T_e \star T} \quad \text{(T-QACallQual)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \textbf{Current} : (!, \bullet, C) \\ \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow T \quad n \geq 0 \\ \Gamma \vdash a_i : T'_i \quad i \in 1..n, \quad \bar{a} = a_1..a_n,\ \ \Gamma \vdash \forall_{i \in 1..n}\ T'_i \preceq T_i\end{array}}{\Gamma \vdash f(\bar{a}) : T} \quad \text{(T-QFCallUnqual)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : T_e,\ \ \Gamma \vdash isControlled(T_e),\ \ \Gamma \vdash ClassType(T_e) = C \\ \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow T \quad n \geq 0 \\ \Gamma \vdash a_i : T'_i \quad i \in 1..n, \quad \bar{a} = a_1..a_n,\ \ \Gamma \vdash \forall_{i \in 1..n}\ T'_i \preceq T_e \otimes T_i\end{array}}{\Gamma \vdash e.f(\bar{a}) : T_e \star T} \quad \text{(T-QFCallQual)}$$

Figure 6.22: Types for expressions

$$\frac{\Gamma \vdash stmts \ \diamond, \ \Gamma \vdash stmt \ \diamond}{\Gamma \vdash stmts; stmt \ \diamond} \quad \text{(T-Seq)}$$

$$\frac{\Gamma \vdash e : (!, \bullet, BOOLEAN), \ \Gamma \vdash stmts \ \diamond, \ \Gamma \vdash stmts' \diamond}{\Gamma \vdash \textbf{if } e \textbf{ then } stmts \textbf{ else } stmts' \textbf{ end } \diamond} \quad \text{(T-If)}$$

$$\frac{\begin{array}{l}\Gamma \vdash T \diamond_{Type}, \ x \notin \Gamma, \ \Gamma, x : T \vdash stmts \ \diamond \\ \Gamma \vdash e : T_e, \ \Gamma \vdash stmts' \ \diamond\end{array}}{\Gamma \vdash \textbf{if } \{x : T\} \ e \textbf{ then } stmts \textbf{ else } stmts' \textbf{ end } \diamond} \quad \text{(T-ObjectTest)}$$

$$\frac{\Gamma \vdash e : (!, \bullet, BOOLEAN), \ \Gamma \vdash stmts \ \diamond, \ \Gamma \vdash stmts' \diamond}{\Gamma \vdash \textbf{from } stmts \textbf{ until } e \textbf{ loop } stmts' \textbf{ end } \diamond} \quad \text{(T-Loop)}$$

$$\frac{\Gamma \vdash isWritable(x), \ \Gamma \vdash x : T_x, \ \Gamma \vdash e : T_e, \ \Gamma \vdash T_e \preceq T_x}{\Gamma \vdash x := e \ \diamond} \quad \text{(T-Assign)}$$

$$\frac{\begin{array}{l}\Gamma \vdash \textbf{Current} : (!, \bullet, C) \\ \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow \varnothing \quad n \geq 0 \\ \Gamma \vdash a_i : T'_i \quad i \in 1..n, \quad \bar{a} = a_1..a_n, \ \Gamma \vdash \forall_{i \in 1..n} \ T'_i \preceq T_i\end{array}}{\Gamma \vdash f(\bar{a}) \ \diamond} \quad \text{(T-CCallUnqual)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : T_e, \ \Gamma \vdash isControlled(T_e), \ \Gamma \vdash ClassType(T_e) = C \\ \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow \varnothing \quad n \geq 0 \\ \Gamma \vdash a_i : T'_i \quad i \in 1..n, \quad \bar{a} = a_1..a_n, \ \Gamma \vdash \forall_{i \in 1..n} \ T'_i \preceq T_e \otimes T_i\end{array}}{\Gamma \vdash e.f(\bar{a}) \ \diamond} \quad \text{(T-CCallQual)}$$

$$\frac{\begin{array}{l}\Gamma \vdash isWritable(x) \\ \Gamma \vdash x : T_x, \ \Gamma \vdash ClassType(T_x) = C, \ \Gamma \vdash f \in Creator(C) \\ \Gamma \vdash FeatureType(C, f) = T_1 \times ... \times T_n \longrightarrow \varnothing \quad n \geq 0 \\ \Gamma \vdash a_i : T'_i \quad i \in 1..n, \quad \bar{a} = a_1..a_n, \ \Gamma \vdash \forall_{i \in 1..n} \ T'_i \preceq T_e \otimes T_i\end{array}}{\Gamma \vdash \textbf{create } x.f(\bar{a}) \ \diamond} \quad \text{(T-Create)}$$

Figure 6.23: Types for statements

$$\frac{\begin{array}{l} rDecl = r\ (x_1 : T_1, ..., x_k : T_k) \\ \qquad \textbf{require}\ pre\ \textbf{local}\ l_1 : Tl_1, ..., l_l : Tl_l\ \textbf{do}\ stmts\ \textbf{ensure}\ post\ \textbf{end} \\ \Gamma \vdash \forall j \in \{1..k\}.\ T_j \diamond_{Type},\ \ \Gamma \vdash \forall j \in \{1..l\}.\ Tl_j \diamond_{Type} \\ \Gamma' = \Gamma, x_1 : T_1, ..., x_k : T_k \\ \Gamma' \vdash pre : (!, \bullet, BOOLEAN),\ \ \Gamma' \vdash post : (!, \bullet, BOOLEAN) \\ \Gamma', l_1 : Tl_1, ..., l_l : Tl_l \vdash stmts \diamond \end{array}}{\Gamma \vdash rDecl : T_1 \times ... \times T_k \longrightarrow \varnothing}$$

(WF-Procedure)

$$\frac{\begin{array}{l} rDecl = r\ (x_1 : T_1, ..., x_k : T_k) : T_{res} \\ \qquad \textbf{require}\ pre\ \textbf{local}\ l_1 : Tl_1, ..., l_l : Tl_l\ \textbf{do}\ stmts\ \textbf{ensure}\ post\ \textbf{end} \\ \Gamma \vdash \forall j \in \{1..k\}.\ T_j \diamond_{Type},\ \ \Gamma \vdash \forall j \in \{1..l\}.\ Tl_j \diamond_{Type},\ \ \Gamma \vdash T_{res}\ \diamond Type \\ \Gamma' = \Gamma, x_1 : T_1, ..., x_k : T_k \\ \Gamma' \vdash pre : (!, \bullet, BOOLEAN),\ \ \Gamma', \textbf{Result} : T_{res} \vdash post : (!, \bullet, BOOLEAN) \\ \Gamma', l_1 : Tl_1, ..., l_l : Tl_l, \textbf{Result} : T_{res} \vdash stmts \diamond \end{array}}{\Gamma \vdash rDecl : T_1 \times ... \times T_k \longrightarrow T_{res}}$$

(WF-Function)

$$\frac{\begin{array}{l} k, l, m, n \geq 0,\ \ \Gamma \vdash \Gamma \diamond \\ \Gamma(C) = \textbf{class}\ C\ \textbf{inherit}\ C'\ \textbf{create}\ cr_1, ..., cr_k \\ \qquad \textbf{feature}\ pt_1, ..., pt_l, a_1 : T_1, ..., a_m : T_m,\ r_1 : RT_1, ...,\ r_n : RT_n\ \ \textbf{end} \\ cDecl = ...\textbf{class}\ C\ \textbf{inherit}\ C'\ \textbf{create}\ cr_1, ..., cr_k \\ \qquad \textbf{feature}\ pt_1, ..., pt_l, a_1 : T_1, ..., a_m : T_m,\ rDecl_1, ...,\ rDecl_n \\ \qquad \textbf{invariant}\ inv\ \textbf{end} \\ \Gamma, \textbf{Current} : (!, \bullet, C) \vdash inv : (!, \bullet, BOOLEAN) \\ \Gamma, \textbf{Current} : (!, \bullet, C) \vdash \forall j \in \{1..m\}.T_j \diamond_{Type} \\ \Gamma, \textbf{Current} : (!, \bullet, C) \vdash \forall j \in \{1..n\}.\ rDecl_j : RT_j \end{array}}{\Gamma \vdash cDecl \diamond}$$

(WF-Class)

$$\frac{\begin{array}{l} n \geq 0,\ \ p = cDecl_1, ..., cDecl_n, RC.r \\ \forall i, j \in \{1..n\}.\ cDecl_i = ...\textbf{class}\ C\ \textbf{inherit}... \ \wedge\ cDecl_j = ...\textbf{class}\ C\ \textbf{inherit}... \implies i = j \\ \Gamma \vdash \forall j \in \{1..n\}.\ cDecl_j \diamond \\ Class(\Gamma) = Class(p) \cup \{ANY, NONE\} \\ \Gamma \vdash RC \in Class(\Gamma),\ \ \Gamma \vdash r \in Creator(RC) \end{array}}{\Gamma \vdash p \diamond}$$

(WF-Program)

Figure 6.24: Well-formedness of routines, classes, and programs

```
p =
    class CLIENT inherit ANY create make
    feature
        buffer : (!, ⊤, BUFFER)

        make
            require True
            do
                create buffer.make_empty
                store (buffer)
                consume (buffer)
            ensure True
            end

        store (a_buffer : (!, ⊤, BUFFER))
            require True
            local b : (?, < a_buffer.handler >, BUFFER)
            do
                create b.make_empty
                a_buffer.put (b)
            ensure not a_buffer.is_empty
            end

        consume (a_buffer : (!, ⊤, BUFFER))
            require not a_buffer.is_empty
            local a : (?, ⊤, ANY)
            do
                a := a_buffer.item
            ensure True
            end

    invariant True
    end
    ...
```

Figure 6.25: Client–buffer program

```
...
class BUFFER inherit ANY create make_empty, put
feature
    item : (?, •, ANY)
    is_empty : (!, •, BOOLEAN)

    make_empty
        require True
        do
            is_empty := True
        ensure True
        end

    put (an_item : (?, •, ANY))
        require True
        do
            item := an_item
            is_empty := False
        ensure True
        end

invariant True
end

expanded class BOOLEAN inherit ANY invariant True end

{CLIENT}.make
```

Figure 6.26: Client–buffer program (cont.)

Γ_p =

class ANY **end**,
class $NONE$ **end**,
class $CLIENT$ **inherit** ANY **create** $make$
feature
$buffer : (!, \top, BUFFER)$, $make : \varnothing \longrightarrow \varnothing$,
$store : (!, \top, BUFFER) \longrightarrow \varnothing$, $consume : (!, \top, BUFFER) \longrightarrow \varnothing$
end,
class $BUFFER$ **inherit** ANY **create** $make\_empty, put$
feature
$item : (?, \bullet, ANY)$, $is\_empty : (!, \bullet, BOOLEAN)$,
$make\_empty : \varnothing \longrightarrow \varnothing$, $put : (?, \bullet, ANY) \longrightarrow \varnothing$
end,
class $BOOLEAN$ **inherit** ANY **end**

Figure 6.27: Typing environment Γ_p

For program p, we construct the typing environment Γ_p given in figure 6.27; we establish

$$Class(p) = \{BOOLEAN, BUFFER, CLIENT\} \tag{6.12.1}$$
$$Class(\Gamma_p) = \{ANY, BOOLEAN, BUFFER, CLIENT, NONE\} \tag{6.12.2}$$

It is easy to demonstrate the well-formedness of Γ_p, i.e.

$$\Gamma_p \vdash \Gamma_p \diamond \tag{6.12.3}$$

by establishing $\sqsubseteq$ and $\preceq$ for Γ_p, showing their acyclicity, and using **WF-EmptyEnv** then **WF-ClassEnv** for each class in $Class(\Gamma_p)$, taking $\Gamma = \Gamma_p$.

To demonstrate that p is correct, we derive $\Gamma_p \vdash p \diamond$ using **WF-Program**; this requires proving the uniqueness and the well-formedness of class declarations in p, the completeness of p, i.e. that p provides declarations for all classes in $Class(\Gamma_p)$ except ANY and $NONE$, and the well-definedness of root class and root creation procedure.

$$\frac{\begin{array}{l}\forall C, C' \in \{BOOLEAN, BUFFER, CLIENT\}. \\ \ cDecl_C = ...\textbf{class}\ C\ \textbf{inherit}... \ \wedge\ cDecl_{C'} = ...\textbf{class}\ C'\ \textbf{inherit}... \implies C = C' \\ Class(\Gamma_p) = Class(p) \cup \{ANY, NONE\} \\ \Gamma_p \vdash \textbf{class}\ BOOLEAN\ \textbf{inherit}\ ANY\ \textbf{invariant}\ True\ \textbf{end} \diamond \\ \Gamma_p \vdash \textbf{class}\ BUFFER\ \textbf{inherit}\ ANY\ ...\ \textbf{invariant}\ True\ \textbf{end} \diamond \\ \Gamma_p \vdash \textbf{class}\ CLIENT\ \textbf{inherit}\ ANY\ ...\ \textbf{invariant}\ True\ \textbf{end} \diamond \\ \Gamma_p \vdash CLIENT \in Class(\Gamma_p) \\ \Gamma_p \vdash make \in Creator(CLIENT)\end{array}}{\Gamma_p \vdash p \diamond}$$

(WF-Class)

To simplify the proof, we will use the following lemma:

Lemma 6.1 (Reflexivity of $\preceq$) *Relation $\preceq$ defined for Γ_p is reflexive, i.e. $\Gamma_p \vdash T \preceq T$ for all T such that $\Gamma_p \vdash T \diamond_{Type}$.*

Proof: Straightforward, by **S-Subclass** followed by **C-Class**, **C-Any**, or **C-None**, depending on $ClassType(T)$. We omit the details here.

We establish now the antecedents of **WF-Program**:

Uniqueness of class declarations

$\forall C, C' \in \{BOOLEAN, BUFFER, CLIENT\}.$
$cDecl_C = ...\textbf{class}\ C\ \textbf{inherit}... \ \wedge\ cDecl_{C'} = ...\textbf{class}\ C'\ \textbf{inherit}... \implies C = C'$
/ immediate from the definition of p /

Completeness

$Class(\Gamma_p) = Class(p) \cup \{ANY, NONE\}$
/ from (6.12.2) and (6.12.1) /

Well-definedness of root class

$\Gamma_p \vdash CLIENT \in Class(\Gamma_p)$
/ from (6.12.2) /

Well-definedness of root creation procedure

$\Gamma_p \vdash make \in Creator(CLIENT)$
/ from the definition of p and the definition of $Creator$ /

Well-formedness of class $BOOLEAN$

$\Gamma_p \vdash \textbf{class}\ BOOLEAN\ \textbf{inherit}\ ANY\ \textbf{invariant}\ True\ \textbf{end}\ \diamond$
/ from **WF-Class**, **T-True**, and (6.12.3) **/**

$$\frac{\Gamma_p \vdash \Gamma_p\ \diamond,\quad \overline{\Gamma_p, \textbf{Current} : (!, \bullet, BOOLEAN) \vdash True : (!, \bullet, BOOLEAN)}\ \text{(T-True)}}{\Gamma_p \vdash \textbf{class}\ BOOLEAN\ \textbf{inherit}\ ANY\ \textbf{invariant}\ True\ \textbf{end}\ \diamond}\ \text{(WF-Class)}$$

Well-formedness of class $BUFFER$

$\Gamma_p \vdash \textbf{class}\ BUFFER\ \textbf{inherit}\ ANY\ ...\ \textbf{invariant}\ True\ \textbf{end}\ \diamond$
/ from **WF-Class /**
Let $\Gamma_b = \Gamma_p, \textbf{Current} : (!, \bullet, BUFFER)$

$$\frac{\begin{array}{l}\Gamma_b \vdash True : (!, \bullet, BOOLEAN)\\ \Gamma_b \vdash (?, \bullet, ANY)\ \diamond_{Type}\\ \Gamma_b \vdash (!, \bullet, BOOLEAN)\ \diamond_{Type}\\ \Gamma_b \vdash make\_empty\ ...\ \textbf{end} : \varnothing \longrightarrow \varnothing\\ \Gamma_b \vdash put(an\_item : (?, \bullet, ANY))\ ...\ \textbf{end} : (?, \bullet, ANY) \longrightarrow \varnothing\end{array}}{\Gamma_p \vdash \textbf{class}\ BUFFER\ \textbf{inherit}\ ANY\ ...\ \textbf{invariant}\ True\ \textbf{end}\ \diamond}\ \text{(WF-Class)}$$

We establish the antecedents of **WF-Class**:

- *Well-definedness of invariant:*
  $\Gamma_b \vdash True : (!, \bullet, BOOLEAN)$
  / from **T-True /**

$$\overline{\Gamma_b \vdash True : (!, \bullet, BOOLEAN)}\ \text{(T-True)}$$

- *Well-definedness of attribute* $item$*:*
  $\Gamma_b \vdash (?, \bullet, ANY)\ \diamond_{Type}$
  / from **WF-ReferenceType**, **A-isExpanded**, **WF-BulletPTag**, **WF-ClassType**, **C-Any /**

$$\overline{\Gamma_b \vdash \bullet\ \diamond_{ProcTag}}\qquad \text{(WF-BulletPTag)}$$

$$\frac{\overline{\Gamma_b \vdash ANY \sqsubseteq ANY}\qquad \text{(C-Any)}}{\Gamma_b \vdash ANY\ \diamond_{ClassType}}\qquad \text{(WF-ClassType)}$$

$$\Gamma_b \vdash \neg isExpanded(ANY) \qquad \text{(A-isExpanded)}$$

$$\frac{? \in \{!, ?\},\ \ \Gamma_b \vdash \bullet \diamond_{ProcTag} \quad \Gamma_b \vdash ANY \diamond_{ClassType},\ \ \Gamma_b \vdash \neg isExpanded(ANY)}{\Gamma_b \vdash (?, \bullet, ANY) \diamond_{Type}} \text{(WF-ReferenceType)}$$

- *Well-definedness of attribute* is_empty*:*
  $\Gamma_b \vdash (!, \bullet, BOOLEAN) \diamond_{Type}$
  / from **WF-ExpandedType**, **A-isExpanded**, **WF-ClassType**, **C-Class /**

$$\frac{\dfrac{\Gamma_b = \Gamma', \textbf{class}\ BOOLEAN\ \textbf{inherit}\ ANY\ ...\ \textbf{end}, \Gamma'' \qquad \text{(definition of } \Gamma_b)}{\Gamma_b \vdash BOOLEAN \sqsubseteq BOOLEAN} \text{(C-Class)}}{\Gamma_b \vdash BOOLEAN \diamond_{ClassType}} \text{(WF-ClassType)}$$

$$\Gamma_b \vdash isExpanded(BOOLEAN) \qquad \text{(A-isExpanded)}$$

$$\frac{\Gamma_b \vdash BOOLEAN \diamond_{ClassType},\ \ \Gamma_b \vdash isExpanded(BOOLEAN)}{\Gamma_b \vdash (!, \bullet, BOOLEAN) \diamond_{Type}} \text{(WF-ExpandedType)}$$

- *Well-definedness of routine* $make\_empty$*:*
  $\Gamma_b \vdash make\_empty\ ...\ \textbf{end} : \varnothing \longrightarrow \varnothing$
  / from **WF-Procedure**, **T-True**, **A-isWritable**, **T-Assign**, **T-QACallUnqual /**

  precondition of $make\_empty$:

$$\frac{}{\Gamma_b \vdash True : (!, \bullet, BOOLEAN)} \text{(T-True)}$$

  postcondition of $make\_empty$: ***idem***

  body of $make\_empty$:

$$\Gamma_b \vdash isWritable(is\_empty) \qquad \text{(A-isWritable)}$$

$$\frac{}{\Gamma_b \vdash True : (!, \bullet, BOOLEAN)} \quad \text{(T-True)}$$

$$\Gamma_b \vdash (!, \bullet, BOOLEAN) \preceq (!, \bullet, BOOLEAN) \qquad \text{(lemma 6.1)}$$

$$\frac{\Gamma_b(\mathbf{Current}) = (!, \bullet, BUFFER)}{\Gamma_b \vdash \mathbf{Current} : (!, \bullet, BUFFER)} \qquad \text{(T-Var)}$$

$$\Gamma_b \vdash FeatureType(BUFFER, is\_empty) = (!, \bullet, BOOLEAN) \quad \text{(def. of } FeatureType\text{)}$$

$$\frac{\begin{array}{l}\Gamma_b \vdash \mathbf{Current} : (!, \bullet, BUFFER) \\ \Gamma_b \vdash FeatureType(BUFFER, is\_empty) = (!, \bullet, BOOLEAN)\end{array}}{\Gamma_b \vdash is\_empty : (!, \bullet, BOOLEAN)} \quad \text{(T-QACallUnqual)}$$

$$\frac{\begin{array}{l}\Gamma_b \vdash isWritable(is\_empty),\ \ \Gamma_b \vdash is\_empty : (!, \bullet, BOOLEAN) \\ \Gamma_b \vdash True : (!, \bullet, BOOLEAN) \\ \Gamma_b \vdash (!, \bullet, BOOLEAN) \preceq (!, \bullet, BOOLEAN)\end{array}}{\Gamma_b \vdash is\_empty\ :=\ True\ \diamond} \qquad \text{(T-Assign)}$$

declaration of $make\_empty$:

$$\frac{\begin{array}{l}\Gamma_b \vdash True : (!, \bullet, BOOLEAN),\ \ \Gamma_b \vdash True : (!, \bullet, BOOLEAN) \\ \Gamma_b \vdash is\_empty\ :=\ True\ \diamond\end{array}}{\Gamma_b \vdash make\_empty\ \mathbf{require}\ True\ \mathbf{do}\ is\_empty\ :=\ True\ \mathbf{ensure}\ True\ \mathbf{end} : \varnothing \longrightarrow \varnothing} \quad \text{(WF-Procedure)}$$

- *Well-definedness of routine* put*:*
  $\Gamma_b \vdash put(an\_item : (?, \bullet, ANY))\ ...\ \mathbf{end} : (?, \bullet, ANY) \longrightarrow \varnothing$
  / from **WF-Procedure**, **T-True**, **T-Seq**, **T-Assign /**
  Let $\Gamma_{bp} = \Gamma_b, an\_item : (?, \bullet, ANY)$

  precondition of put:

$$\frac{}{\Gamma_{bp} \vdash True : (!, \bullet, BOOLEAN)} \text{ (T-True)}$$

postcondition of put:

$$\frac{}{\Gamma_{bp} \vdash True : (!, \bullet, BOOLEAN)} \text{ (T-True)}$$

body of put:

$$\Gamma_{bp} \vdash isWritable(item) \qquad \text{(A-isWritable)}$$

$$\frac{\Gamma_{bp}(\mathbf{Current}) = (!, \bullet, BUFFER)}{\Gamma_{bp} \vdash \mathbf{Current} : (!, \bullet, BUFFER)} \text{ (T-Var)}$$

$$\Gamma_{bp} \vdash FeatureType(BUFFER, item) = (?, \bullet, ANY) \qquad \text{(def. of } FeatureType\text{)}$$

$$\frac{\Gamma_{bp} \vdash \mathbf{Current} : (!, \bullet, BUFFER) \quad \Gamma_{bp} \vdash FeatureType(BUFFER, item) = (?, \bullet, ANY)}{\Gamma_{bp} \vdash item : (?, \bullet, ANY)} \text{ (T-QACallUnqual)}$$

$$\frac{\Gamma_{bp}(an\_item) = (?, \bullet, ANY)}{\Gamma_{bp} \vdash an\_item : (?, \bullet, ANY)} \text{ (T-Var)}$$

$$\Gamma_{bp} \vdash (?, \bullet, ANY) \preceq (?, \bullet, ANY) \qquad \text{(lemma 6.1)}$$

$$\frac{\Gamma_{bp} \vdash isWritable(item), \;\; \Gamma_{bp} \vdash item : (?, \bullet, ANY) \quad \Gamma_{bp} \vdash an\_item : (?, \bullet, ANY) \quad \Gamma_{bp} \vdash (?, \bullet, ANY) \preceq (?, \bullet, ANY)}{\Gamma_{bp} \vdash item \;:=\; an\_item \diamond} \text{ (T-Assign)}$$

$$\Gamma_{bp} \vdash isWritable(is\_empty) \qquad \text{(A-isWritable)}$$

$$\frac{}{\Gamma_{bp} \vdash False : (!, \bullet, BOOLEAN)} \text{ (T-False)}$$

$$\Gamma_{bp} \vdash (!, \bullet, BOOLEAN) \preceq (!, \bullet, BOOLEAN) \qquad \text{(lemma 6.1)}$$

$$\Gamma_{bp} \vdash FeatureType(BUFFER, is\_empty) = (!, \bullet, BOOLEAN) \quad \text{(def. of } FeatureType\text{)}$$

$$\frac{\Gamma_{bp} \vdash \textbf{Current} : (!, \bullet, BUFFER) \quad \Gamma_{bp} \vdash FeatureType(BUFFER, is\_empty) = (!, \bullet, BOOLEAN)}{\Gamma_{bp} \vdash is\_empty : (!, \bullet, BOOLEAN)} \text{ (T-QACallUnqual)}$$

$$\frac{\Gamma_{bp} \vdash isWritable(is\_empty), \ \Gamma_{bp} \vdash is\_empty : (!, \bullet, BOOLEAN) \quad \Gamma_{bp} \vdash False : (!, \bullet, BOOLEAN) \quad \Gamma_{bp} \vdash (!, \bullet, BOOLEAN) \preceq (!, \bullet, BOOLEAN)}{\Gamma_{bp} \vdash is\_empty \ := \ False \diamond} \text{ (T-Assign)}$$

$$\frac{\Gamma_{bp} \vdash item \ := \ an\_item \diamond, \ \Gamma_{bp} \vdash is\_empty \ := \ False \diamond}{\Gamma_{bp} \vdash item \ := \ an\_item; \ is\_empty \ := \ False \diamond} \text{ (T-Seq)}$$

declaration of put:

$$\frac{\Gamma_b \vdash True : (!, \bullet, BOOLEAN), \ \Gamma_b \vdash True : (!, \bullet, BOOLEAN) \quad \Gamma_{bp} \vdash item \ := \ an\_item; \ is\_empty \ := \ False \diamond}{\begin{array}{l}\Gamma_b \vdash put \ (an\_item : (?, \bullet, ANY)) \ \textbf{require} \ True \\ \qquad \textbf{do} \ item \ := \ an\_item; \ is\_empty \ := \ False \\ \qquad \textbf{ensure} \ True \ \textbf{end} : (?, \bullet, ANY) \longrightarrow \varnothing\end{array}} \text{ (WF-Procedure)}$$

Well-formedness of class $CLIENT$

$\Gamma_p \vdash$ **class** $CLIENT$ **inherit** ANY ... **invariant** $True$ **end** $\diamond$
/ from **WF-Class /**

Let $\Gamma_c = \Gamma_p, \mathbf{Current} : (!, \bullet, CLIENT)$

$$\frac{\begin{array}{l}\Gamma_c \vdash True : (!, \bullet, BOOLEAN) \\ \Gamma_c \vdash (!, \top, BUFFER) \diamond_{Type} \\ \Gamma_c \vdash make \ ... \ \mathbf{end} : \varnothing \longrightarrow \varnothing \\ \Gamma_c \vdash store(a\_buffer : (!, \top, BUFFER)) \ ... \ \mathbf{end} : (!, \top, BUFFER) \longrightarrow \varnothing \\ \Gamma_c \vdash consume(a\_buffer : (!, \top, BUFFER)) \ ... \ \mathbf{end} : (!, \top, BUFFER) \longrightarrow \varnothing\end{array}}{\Gamma_p \vdash \mathbf{class}\ CLIENT\ \mathbf{inherit}\ ANY\ ...\ \mathbf{invariant}\ True\ \mathbf{end} \diamond}$$ (WF-Class)

We establish the antecedents of **WF-Class**:

- *Well-definedness of invariant:*
  $\Gamma_c \vdash True : (!, \bullet, BOOLEAN)$
  / from **T-True /**

$$\frac{}{\Gamma_c \vdash \mathbf{True} : (!, \bullet, BOOLEAN)}$$ (T-True)

- *Well-definedness of attribute* $buffer$*:*
  $\Gamma_c \vdash (!, \top, BUFFER) \diamond_{Type}$
  / from **WF-ReferenceType**, **A-isExpanded**, **WF-ClassType**, **C-Class**, **WF-TopPTag /**

$$\frac{}{\Gamma_c \vdash \top \diamond_{ProcTag}}$$ (WF-TopPTag)

$$\frac{\Gamma_c = \Gamma', \mathbf{class}\ BUFFER\ \mathbf{inherit}\ ANY\ ...\mathbf{end}, \Gamma''}{\Gamma_c \vdash BUFFER \sqsubseteq BUFFER}\ \text{(definition of } \Gamma_c\text{)}\ \text{(C-Class)}$$
$$\frac{\Gamma_c \vdash BUFFER \sqsubseteq BUFFER}{\Gamma_c \vdash BUFFER \diamond_{ClassType}}\ \text{(C-Class)}\ \text{(WF-ClassType)}$$

$$\Gamma_c \vdash \neg isExpanded(BUFFER)$$ (A-isExpanded)

$$\frac{! \in \{!, ?\},\ \Gamma_c \vdash \top \diamond_{ProcTag} \quad \Gamma_c \vdash BUFFER \diamond_{ClassType},\ \Gamma_c \vdash \neg isExpanded(BUFFER)}{\Gamma_c \vdash (!, \top, BUFFER) \diamond_{Type}}$$ (WF-ReferenceType)

- *Well-definedness of routine* $make$*:*
  $\Gamma_c \vdash make$ ... $\mathbf{end} : \varnothing \longrightarrow \varnothing$
  / from **WF-Procedure**, **T-True**, **T-Seq**, **T-Create**, **T-CCallUnqual /**

  precondition of $make$:

$$\frac{}{\Gamma_c \vdash True : (!, \bullet, BOOLEAN)} \text{ (T-True)}$$

  postcondition of $make$: ***idem***

  body of $make$:

$$\frac{\Gamma_c(\mathbf{Current}) = (!, \bullet, CLIENT)}{\Gamma_c \vdash \mathbf{Current} : (!, \bullet, CLIENT)} \text{ (T-Var)}$$

$$\Gamma_c \vdash (!, \top, BUFFER) \preceq (!, \top, BUFFER) \quad \text{(lemma 6.1)}$$

$$\Gamma_c \vdash FeatureType(CLIENT, store) = (!, \top, BUFFER) \longrightarrow \varnothing \text{ (def. of } FeatureType)$$

$$\Gamma_c \vdash FeatureType(CLIENT, consume) = (!, \top, BUFFER) \longrightarrow \varnothing \text{ (def. of } FeatureType)$$

$$\Gamma_c \vdash FeatureType(BUFFER, make\_empty) = \varnothing \longrightarrow \varnothing \quad \text{(def. of } FeatureType)$$

$$\Gamma_c \vdash FeatureType(CLIENT, buffer) = (!, \top, BUFFER) \longrightarrow \varnothing \text{ (def. of } FeatureType)$$

$$\frac{\Gamma_c \vdash \mathbf{Current} : (!, \bullet, CLIENT) \quad \Gamma_c \vdash FeatureType(CLIENT, buffer) = (!, \top, BUFFER)}{\Gamma_c \vdash buffer : (!, \top, BUFFER)} \text{ (T-QACallUnqual)}$$

$$\Gamma_c \vdash ClassType((!, \top, BUFFER)) = BUFFER \quad \text{(A-ClassType)}$$

$$\Gamma_c \vdash make\_empty \in Creator(BUFFER) \quad \text{(definition of } Creator)$$

$$\Gamma_c \vdash isWritable(buffer) \qquad \text{(A-isWritable)}$$

$$\frac{\begin{array}{l}\Gamma_c \vdash isWritable(buffer),\ \ \Gamma_c \vdash buffer : (!, \top, BUFFER) \\ \Gamma_c \vdash ClassType((!, \top, BUFFER)) = BUFFER \\ \Gamma_c \vdash make\_empty \in Creator(BUFFER) \\ \Gamma_c \vdash FeatureType(BUFFER, make\_empty) = \varnothing \longrightarrow \varnothing\end{array}}{\Gamma_c \vdash \textbf{create}\ buffer.make\_empty \diamond} \qquad \text{(T-Create)}$$

$$\frac{\begin{array}{l}\Gamma_c \vdash \textbf{Current} : (!, \bullet, CLIENT) \\ \Gamma_c \vdash FeatureType(CLIENT, store) = (!, \top, BUFFER) \\ \Gamma_c \vdash buffer : (!, \top, BUFFER),\ \ \Gamma_c \vdash (!, \top, BUFFER) \preceq (!, \top, BUFFER)\end{array}}{\Gamma_c \vdash store(buffer) \diamond}$$

(T-CCallUnqual)

$$\frac{\begin{array}{l}\Gamma_c \vdash \textbf{Current} : (!, \bullet, CLIENT) \\ \Gamma_c \vdash FeatureType(CLIENT, consume) = (!, \top, BUFFER) \\ \Gamma_c \vdash buffer : (!, \top, BUFFER),\ \ \Gamma_c \vdash (!, \top, BUFFER) \preceq (!, \top, BUFFER)\end{array}}{\Gamma_c \vdash consume(buffer) \diamond}$$

(T-CCallUnqual)

$$\frac{\Gamma_c \vdash \textbf{create}\ buffer.make\_empty \diamond,\ \ \Gamma_c \vdash store(buffer) \diamond}{\Gamma_c \vdash \textbf{create}\ buffer.make\_empty; store(buffer) \diamond} \qquad \text{(T-Seq)}$$

$$\frac{\begin{array}{l}\Gamma_c \vdash \textbf{create}\ buffer.make\_empty; store(buffer) \diamond \\ \Gamma_c \vdash consume(buffer) \diamond\end{array}}{\Gamma_c \vdash \textbf{create}\ buffer.make\_empty; store(buffer); consume(buffer) \diamond} \qquad \text{(T-Seq)}$$

declaration of $make$:

$$\frac{\begin{array}{l}\Gamma_c \vdash True : (!, \bullet, BOOLEAN),\ \ \Gamma_c \vdash True : (!, \bullet, BOOLEAN) \\ \Gamma_c \vdash \textbf{create}\ buffer.make\_empty; store(buffer); consume(buffer) \diamond\end{array}}{\begin{array}{l}\Gamma_c \vdash make\ \textbf{require}\ True \\ \qquad \textbf{do create}\ buffer.make\_empty; store(buffer); consume(buffer) \\ \qquad \textbf{ensure}\ True\ \textbf{end} : \varnothing \longrightarrow \varnothing\end{array}}$$

(WF-Procedure)

- *Well-definedness of routine store:*
  $\Gamma_c \vdash store(a\_buffer : (!, \top, BUFFER))$ ... **end** $: (!, \top, BUFFER) \longrightarrow \varnothing$
  / from **WF-Procedure**, **T-True**, **T-Seq**, **T-Create**, **T-CCallQual /**
  Let $\Gamma_{cs} = \Gamma_c, a\_buffer : (!, \top, BUFFER)$
  Let $\Gamma_{csb} = \Gamma_{cs}, b : (!, < a\_buffer.\mathbf{handler} >, BUFFER)$

  precondition of $store$:

$$\frac{}{\Gamma_{cs} \vdash True : (!, \bullet, BOOLEAN)} \text{ (T-True)}$$

  postcondition of $store$:

$$\frac{\Gamma_{cs}(a\_buffer) = (!, \top, BUFFER)}{\Gamma_{cs} \vdash a\_buffer : (!, \top, BUFFER)} \text{ (T-Var)}$$

$$\Gamma_{cs} \vdash \neg isWritable(a\_buffer) \quad \text{(A-isWritable)}$$

$$\frac{\Gamma_{cs} \vdash a\_buffer : (!, \top, BUFFER), \;\; \Gamma_{cs} \vdash \neg isWritable(a\_buffer)}{\Gamma_{cs} \vdash a\_buffer : (!, < a\_buffer.\mathbf{handler} >, BUFFER)} \text{ (T-Implicit)}$$

$$\Gamma_{cs} \vdash FeatureType(BUFFER, is\_empty) = (!, \bullet, BOOLEAN) \quad \text{(def. of } FeatureType\text{)}$$

$$\frac{\begin{array}{l}\Gamma_{cs} \vdash a\_buffer : (!, < a\_buffer.\mathbf{handler} >, BUFFER) \\ \Gamma_{cs} \vdash isControlled(a\_buffer) \\ \Gamma_{cs} \vdash ClassType((!, < a\_buffer.\mathbf{handler} >, BUFFER)) = BUFFER \\ \Gamma_{cs} \vdash FeatureType(BUFFER, is\_empty) = (!, \bullet, BOOLEAN)\end{array}}{\Gamma_{cs} \vdash a\_buffer.is\_empty : (!, < a\_buffer.\mathbf{handler} >, BUFFER) \star (!, \bullet, BOOLEAN)} \text{ (T-QACallQual)}$$

$$\Gamma_{cs} \vdash (!, < a\_buffer.\mathbf{handler} >, BUFFER) \star (!, \bullet, BOOLEAN) = (!, \bullet, BOOLEAN) \quad \text{(TC-}\star\text{)}$$

$$\frac{\Gamma_{cs} \vdash a\_buffer.is\_empty : (!, \bullet, BOOLEAN)}{\Gamma_{cs} \vdash \mathbf{not}\ a\_buffer.is\_empty : (!, \bullet, BOOLEAN)} \text{ (T-Not)}$$

body of *store*:

$$\frac{\Gamma_{csb}(b) = (!, < a\_buffer.\mathbf{handler} >, BUFFER)}{\Gamma_{csb} \vdash b : (!, < a\_buffer.\mathbf{handler} >, BUFFER)} \quad \text{(T-Var)}$$

$$\Gamma_{csb} \vdash isWritable(b) \qquad \text{(A-isWritable)}$$

$$\Gamma_{csb} \vdash ClassType((!, < a\_buffer.\mathbf{handler} >, BUFFER)) = BUFFER \quad \text{(A-ClassType)}$$

$$\Gamma_{csb} \vdash make\_empty \in Creator(BUFFER) \qquad \text{(definition of } Creator\text{)}$$

$$\Gamma_{csb} \vdash FeatureType(BUFFER, make\_empty) = \varnothing \longrightarrow \varnothing \quad \text{(def. of } FeatureType\text{)}$$

$$\frac{\begin{array}{l}\Gamma_{csb} \vdash isWritable(b), \;\; \Gamma_{csb} \vdash b : (!, < a\_buffer.\mathbf{handler} >, BUFFER) \\ \Gamma_c \vdash ClassType((!, < a\_buffer.\mathbf{handler} >, BUFFER)) = BUFFER \\ \Gamma_c \vdash make\_empty \in Creator(BUFFER) \\ \Gamma_{csb} \vdash FeatureType(BUFFER, make\_empty) = \varnothing \longrightarrow \varnothing\end{array}}{\Gamma_{csb} \vdash \mathbf{create}\ b.make\_empty \diamond} \quad \text{(T-Create)}$$

$$\frac{\Gamma_{csb}(a\_buffer) = (!, \top, BUFFER)}{\Gamma_{csb} \vdash a\_buffer : (!, \top, BUFFER)} \quad \text{(T-Var)}$$

$$\Gamma_{csb} \vdash \neg isWritable(a\_buffer) \qquad \text{(A-isWritable)}$$

$$\frac{\Gamma_{csb} \vdash a\_buffer : (!, \top, BUFFER), \;\; \Gamma_{csb} \vdash \neg isWritable(a\_buffer)}{\Gamma_{csb} \vdash a\_buffer : (!, < a\_buffer.\mathbf{handler} >, BUFFER)} \quad \text{(T-Implicit)}$$

$$\Gamma_{csb} \vdash isControlled(a\_buffer) \qquad \text{(A-isControlled)}$$

$$\Gamma_{csb} \vdash FeatureType(BUFFER, put) = (?, \bullet, ANY) \longrightarrow \varnothing \quad \text{(def. of } FeatureType\text{)}$$

$$\frac{\Gamma_{csb} = \Gamma', \textbf{class}\ BUFFER\ \textbf{inherit}\ ANY\ ...\ \textbf{end}, \Gamma''}{\Gamma_{csb} \vdash BUFFER \sqsubseteq ANY}\ \text{(definition of } \Gamma_{csb}\text{)}\quad \text{(C-Subclass)}$$

$$\frac{\Gamma_{csb} \vdash BUFFER \sqsubseteq ANY}{\Gamma_{csb} \vdash (?, < a\_buffer.\textbf{handler} >, BUFFER) \preceq (?, < a\_buffer.\textbf{handler} >, ANY)}\quad \text{(S-Subclass)}$$

$$\frac{\Gamma_{csb} \vdash (?, < a\_buffer.\textbf{handler} >, BUFFER) \preceq (?, < a\_buffer.\textbf{handler} >, ANY)}{\Gamma_{csb} \vdash (!, < a\_buffer.\textbf{handler} >, BUFFER) \preceq (?, < a\_buffer.\textbf{handler} >, ANY)}\quad \text{(S-Attached)}$$

$$\Gamma_{csb} \vdash (!, < a\_buffer.\textbf{handler} >, BUFFER) \otimes (?, \bullet, ANY)) = (?, < a\_buffer.\textbf{handler} >, ANY) \quad \text{(TC-}\otimes\text{)}$$

$$\frac{\begin{array}{l}\Gamma_{csb} \vdash a\_buffer : (!, < a\_buffer.\textbf{handler} >, BUFFER)\\ \Gamma_{csb} \vdash isControlled(a\_buffer)\\ \Gamma_{csb} \vdash ClassType((!, < a\_buffer.\textbf{handler} >, BUFFER)) = BUFFER\\ \Gamma_{csb} \vdash FeatureType(BUFFER, put) = (?, \bullet, ANY) \longrightarrow \varnothing\\ \Gamma_{csb} \vdash b : (!, < a\_buffer.\textbf{handler} >, BUFFER)\\ \Gamma_{csb} \vdash (!, < a\_buffer.\textbf{handler} >, BUFFER) \preceq (!, < a\_buffer.\textbf{handler} >, BUFFER) \otimes (?, \bullet, ANY))\end{array}}{\Gamma_{csb} \vdash a\_buffer.put\ (b)\ \diamond}$$

(T-CCallQual)

$$\frac{\Gamma_{csb} \vdash \textbf{create}\ b.make\_empty\ \diamond,\ \ \Gamma_{csb} \vdash a\_buffer.put\ (b)\ \diamond}{\Gamma_{csb} \vdash \textbf{create}\ b.make\_empty; a\_buffer.put\ (b)\ \diamond}\quad \text{(T-Seq)}$$

declaration of $store$:

$$\Gamma_{cs} \vdash \neg isAttached((!, \top, BUFFER)) \quad \text{(A-isAttached)}$$

$$\frac{\Gamma_{cs} = \Gamma_c, a\_buffer : (!, \top, BUFFER)\quad \Gamma_{cs} \neg isWritable(a\_buffer),\ \ \Gamma_{cs} \vdash isAttached((!, \top, BUFFER))}{\Gamma_{cs} \vdash < a\_buffer.\textbf{handler} >\ diamond_{ProcTag}}$$

(WF-QualifiedPTag)

$$\frac{! \in \{?, !\},\ \ \Gamma_{cs} \vdash < a\_buffer.\textbf{handler} >\ diamond_{ProcTag}\quad \Gamma_{cs} \vdash BUFFER\ \diamond_{ClassType},\ \ \Gamma_{cs} \vdash \neg isExpanded(BUFFER)}{\Gamma_{cs} \vdash (!, < a\_buffer.\textbf{handler} >, BUFFER)\ \diamond_{Type}}$$

(WF-ReferenceType)

$$\frac{\begin{array}{l}\Gamma_{cs} \vdash (!, < a\_buffer.\mathbf{handler} >, BUFFER) \diamond_{Type} \\ \Gamma_{cs} \vdash True : (!, \bullet, BOOLEAN) \\ \Gamma_{cs} \vdash \mathbf{not}\ a\_buffer.is\_empty : (!, \bullet, BOOLEAN) \\ \Gamma_{csb} \vdash \mathbf{create}\ b.make\_empty; a\_buffer.put\ (b) \diamond\end{array}}{\begin{array}{l}\Gamma_c \vdash store\ \mathbf{require}\ True\ \mathbf{local}\ b : (!, < a\_buffer.\mathbf{handler} >, BUFFER) \\ \quad \mathbf{do\ create}\ b.make\_empty; a\_buffer.put\ (b) \\ \quad \mathbf{ensure\ not}\ a\_buffer.is\_empty\ \mathbf{end} : (!, \top, BUFFER) \longrightarrow \varnothing\end{array}}$$

(WF-Procedure)

- *Well-definedness of routine* $consume$*:*
  $\Gamma_c \vdash consume(a\_buffer : (!, \top, BUFFER))$ ... **end** $: (!, \top, BUFFER) \longrightarrow \varnothing$
  / from **WF-Procedure**, **T-True**, **T-Seq**, **T-Create**, **T-CCallQual /**
  Let $\Gamma_{cs} = \Gamma_c, a\_buffer : (!, \top, BUFFER)$
  Let $\Gamma_{csb} = \Gamma_{cs}, a : (?, \top, ANY)$

  precondition of $consume$: proof as for the postcondition of $store$

  postcondition of $consume$: proof as for the precondition of $store$

  body of $consume$:

$$\frac{\Gamma_{csb}(a) = (?, \top, ANY)}{\Gamma_{csb} \vdash a : (?, \top, ANY)} \quad \text{(T-Var)}$$

$$\Gamma_{csb} \vdash isWritable(a) \quad \text{(A-isWritable)}$$

$$\Gamma_{csb} \vdash FeatureType(BUFFER, item) = (?, \bullet, ANY) \quad \text{(def. of } FeatureType\text{)}$$

$$\frac{\Gamma_{csb}(a\_buffer) = (!, \top, BUFFER)}{\Gamma_{csb} \vdash a\_buffer : (!, \top, BUFFER)} \quad \text{(T-Var)}$$

$$\Gamma_{csb} \vdash isControlled(a\_buffer) \quad \text{(A-isControlled)}$$

$$\frac{\begin{array}{l}\Gamma_{csb} \vdash a\_buffer : (!, < a\_buffer.\textbf{handler} >, BUFFER) \\ \Gamma_{csb} \vdash isControlled(a\_buffer) \\ \Gamma_{csb} \vdash ClassType((!, < a\_buffer.\textbf{handler} >, BUFFER)) = BUFFER \\ \Gamma_{csb} \vdash FeatureType(BUFFER, item) = (?, \bullet, ANY)\end{array}}{\Gamma_{csb} \vdash a\_buffer.item : (!, < a\_buffer.\textbf{handler} >, BUFFER) \star (?, \bullet, ANY)} \text{(T-QACallQual)}$$

$$\begin{array}{l}\Gamma_{csb} \vdash (!, < a\_buffer.\textbf{handler} >, BUFFER) \star (?, \bullet, ANY) \\ \qquad = (?, < a\_buffer.\textbf{handler} >, ANY)\end{array} \quad \text{(TC-}\star\text{)}$$

$$\frac{\overline{\Gamma_{csb} \vdash ANY \sqsubseteq ANY} \quad \text{(C-Any)}}{\Gamma_{csb} \vdash (?, < a\_buffer.\textbf{handler} >, ANY) \preceq (?, \top, ANY)} \quad \text{(S-Top)}$$

$$\frac{\begin{array}{l}\Gamma_{csb} \vdash a : (?, \top, ANY), \;\; \Gamma_{csb} \vdash isWritable(a) \\ \Gamma_{csb} \vdash a\_buffer.item : (?, < a\_buffer.\textbf{handler} >, ANY) \\ \Gamma_{csb} \vdash (?, < a\_buffer.\textbf{handler} >, ANY) \preceq (?, \top, ANY)\end{array}}{\Gamma_{csb} \vdash a \;:=\; a\_buffer.item \diamond} \quad \text{(T-Assign)}$$

declaration of $consume$:

$$\frac{}{\Gamma_{cs} \vdash \top \diamond_{ProcTag}} \quad \text{(WF-TopPTag)}$$

$$\frac{\overline{\Gamma_{cs} \vdash ANY \sqsubseteq ANY} \quad \text{(C-Any)}}{\Gamma_{cs} \vdash ANY \diamond_{ClassType}} \quad \text{(WF-ClassType)}$$

$$\Gamma_{cs} \vdash \neg isExpanded(ANY) \quad \text{(A-isExpanded)}$$

$$\frac{\begin{array}{l}? \in \{?, !\}, \;\; \Gamma_{cs} \vdash \top \diamond_{ProcTag} \\ \Gamma_{cs} \vdash ANY \diamond_{ClassType}, \;\; \Gamma_{cs} \vdash \neg isExpanded(ANY)\end{array}}{\Gamma_{cs} \vdash (?, \top, ANY) \diamond_{Type}} \quad \text{(WF-ReferenceType)}$$

$$\frac{\begin{array}{l}\Gamma_{cs} \vdash (?, \top, ANY) \diamond_{Type} \\ \Gamma_{cs} \vdash \textbf{not}\ a\_buffer.is\_empty : (!, \bullet, BOOLEAN) \\ \Gamma_{cs} \vdash True : (!, \bullet, BOOLEAN) \\ \Gamma_{csb} \vdash a\ :=\ a\_buffer.item \diamond\end{array}}{\begin{array}{l}\Gamma_c \vdash consume\ \textbf{require not}\ a\_buffer.is\_empty\ \textbf{local}\ a : (?, \top, ANY) \\ \quad \textbf{do}\ a\ :=\ a\_buffer.item\ \textbf{ensure}\ True\ \textbf{end} : (!, \top, BUFFER) \longrightarrow \varnothing\end{array}}$$

(WF-Procedure)

Well-formedness and completeness of p

$\Gamma_p \vdash p \diamond$
/ straightforward application of **WF-Class**, having proved all the antecedents /

$$\frac{\begin{array}{l}\forall C, C' \in \{BOOLEAN, BUFFER, CLIENT\}. \\ \ cDecl_C = ...\textbf{class}\ C\ \textbf{inherit}... \ \wedge\ cDecl_{C'} = ...\textbf{class}\ C'\ \textbf{inherit}... \implies C = C' \\ Class(\Gamma_p) = Class(p) \cup \{ANY, NONE\} \\ \Gamma_p \vdash \textbf{class}\ BOOLEAN\ \textbf{inherit}\ ANY\ \textbf{invariant}\ True\ \textbf{end} \diamond \\ \Gamma_p \vdash \textbf{class}\ BUFFER\ \textbf{inherit}\ ANY\ ...\ \textbf{invariant}\ True\ \textbf{end} \diamond \\ \Gamma_p \vdash \textbf{class}\ CLIENT\ \textbf{inherit}\ ANY\ ...\ \textbf{invariant}\ True\ \textbf{end} \diamond \\ \Gamma_p \vdash CLIENT \in Class(\Gamma_p) \\ \Gamma_p \vdash make \in Creator(CLIENT)\end{array}}{\Gamma_p \vdash p \diamond}$$

(WF-Class)

This completes the proof of program p. □

6.12.2 Lemmas

This section discusses two important properties ensured by our type system: correct wrapping of separate calls, and monotonicity of separate references. We prove the corresponding lemmas which may be used in a proof of soundness.

Monotonicity of separate references

Multidot expressions involving separate calls must be themselves separate, unless all separate calls are followed by a call to a feature returning an expanded result. This property is essential in proving the absence of traitors, i.e. non-separate expressions that represent separate objects. We formalise it as follows.

Lemma 6.2 (Monotonicity of separate references)
An expression e_n of the form $f_0.f_1.\cdots.f_n,\ n \geq 0$, is separate, i.e.

$$\Gamma \vdash e_n : (\gamma_n, \alpha_n, C_n) \ \wedge \ \alpha_n \notin \{\bullet, \mathbf{Current.handler}\}$$

if $\exists i \in 0..n.$ (f_i is separate $\wedge\ \forall j \geq i.$ f_j is not expanded).

Proof: By induction on the shape of e_n.

Induction basis: We take $i \in \{0..n\}$ such that

$$\Gamma \vdash f_i : (\gamma, \alpha, C) \ \wedge \ \alpha \notin \{\bullet, \mathbf{Current.handler}\}$$
$$\wedge \ \forall j \geq i.(f_j : (\gamma', \alpha', C') \ \wedge \ \neg isExpanded(C') \ \wedge \ \alpha' \in \{\bullet, \mathbf{Current.handler}\})$$

- Case $i = 0$
  $e_i = f_0$, hence $\Gamma \vdash e_i : (\gamma, \alpha, C) \ \wedge \ \alpha \notin \{\bullet, \mathbf{Current.handler}\}$.

- Case $i > 0$
  $e_i = e_{i-1}.f_i$, where $e_{i-1} = f_0.\cdots.f_{i-1} \ \wedge \ \Gamma \vdash e_{i-1} : (\gamma_{i-1}, \alpha_{i-1}, C_{i-1})$
  The type of e_i can be derived only from **C-QACallQual** or **C-QFCallQual**; in both cases we have
  $$\Gamma \vdash e_i : (\gamma_{i-1}, \alpha_{i-1}, C_{i-1}) \star (\gamma, \alpha, C)$$
  From **TC-$\star$** and $\alpha \notin \{\bullet, \mathbf{Current.handler}\}$, we obtain
  $$(\gamma_{i-1}, \alpha_{i-1}, C_{i-1}) \star (\gamma, \alpha, C) = (\gamma, \top, C)$$
  Therefore $\Gamma \vdash e_i : (\gamma, \top, C)$.

Induction step: $e_{j+1} = e_j.f_{j+1}$, $\Gamma \vdash e_j : (\gamma_j, \alpha_j, C_j) \ \wedge \ \alpha_j \notin \{\bullet, \mathbf{Current.handler}\}$, and $\Gamma \vdash f_{j+1} : (\gamma, \alpha, C) \ \wedge \ \alpha \in \{\bullet, \mathbf{Current.handler}\}$.

The type of $e_j + 1$ can be derived only from **C-QACallQual** or **C-QFCallQual**; in both cases we have
$$\Gamma \vdash e_{j+1} : (\gamma_j, \alpha_j, C_j) \star (\gamma, \alpha, C)$$
From **TC-$\star$** and the assumptions $\alpha_j \notin \{\bullet, \mathbf{Current.handler}\}$
and $\alpha \in \{\bullet, \mathbf{Current.handler}\}$, we obtain
$$(\gamma_j, \alpha_j, C_j) \star (\gamma, \alpha, C) = (\gamma, \alpha_j, C)$$
Therefore $\Gamma \vdash e_{j+1} : (\gamma, \alpha_j, C) \ \wedge \ \alpha_j \notin \{\bullet, \mathbf{Current.handler}\}$.

□

Correct wrapping of separate calls

According to the call validity rule 6.5.3, the target of a feature call must be *controlled*. The definition of controllability (6.5.2) states that an expression e is controlled if and only if e is attached and either non-separate or having the same explicit processor tag as some attached formal arguments of the enclosing routine. The attachability requirement eliminates calls on a

void target; the additional conditions ensure a correct locking of the target: the target must be handled either by the current processor (in which case it is trivially locked by the client) or by a processor handling one of the locked arguments (in which case it is locked as well). The type rules given in section 6.11 have been devised in a way that ensures controllability of call targets. We prove the following lemma concerning the controllability of separate targets.

Lemma 6.3 (Correct wrapping of separate calls)
If an expression e is separate, i.e. $\Gamma \vdash e : (\gamma, \alpha, C) \ \wedge \ \alpha \notin \{\bullet, \mathbf{Current.handler}\}$, then a feature call on e may occur only in the precondition, the postcondition, or the body of a routine r which takes an attached formal argument $farg$ such that e and is non-separate from $farg$.

Proof: On the derivation of $e.f$.
There are three rules whose conclusions match $e.f$: **T-CCallQual** ($\Gamma \vdash e.f(...) \diamond$), **T-QACallQual** ($\Gamma \vdash e.f : T$), and **T-QFCallQual** ($\Gamma \vdash e.f(...) : T$).

- Case **T-CCallQual**
  T-CCallQual has an antecedent $isControlled((\gamma, \alpha, C))$. Only one rule allows the derivation of $isControlled((\gamma, \alpha, C))$:

$$\frac{\begin{array}{l}\Gamma \vdash farg : (!, \alpha_{farg}, C_{farg}), \ \ \Gamma \vdash isFormalArgument(farg) \\ \Gamma \vdash C \diamond_{ClassType} \\ \gamma = !, \ \ \Gamma \vdash \alpha = farg.\mathbf{handler}\end{array}}{\Gamma \vdash isControlled((\gamma, \alpha, C))} \quad \text{(A-isControlled)}$$

  From **A-isFormalArgument** we know that either $farg = \mathbf{Current}$ or $farg$ is a formal argument of some routine r.

  - Case $farg = \mathbf{Current}$
    $\alpha = Current.handler$ contradicts the assumption $\alpha \notin \{\bullet, \mathbf{Current.handler}\}$.
  - Case $farg \neq \mathbf{Current}$
    $farg \in dom(\Gamma)$ only in the antecedents of **WF-Procedure** or **WF-Function** applied to r, where Γ is obtained by enriching the environment with declarations of all formal arguments of r. There are three such antecedents: one for the precondition, one for the postcondition, and one for the body of r. Therefore, $e.f(...)$ must occur in one of these contexts.

  The above derivation using **A-isControlled** now permits concluding that:

  - the call $e.f(...)$ occurs in the precondition, the postcondition, or the body of some routine r,
  - e is non-separate from $farg$,
  - $farg$ is a formal argument of r,
  - $farg$ is attached.

- Case **T-QACallQual**: idem
- Case **T-CCallQual**: idem

□

7

Flexible locking

THE access control policy of SCOOP, as described so far, provides strong safety guarantees: the mutual exclusion and the atomicity at the routine level, and the FIFO scheduling of clients' calls. Unfortunately, these guarantees come at a high price: all arguments of a feature call have to be locked, even if they are never used by the feature. In most cases, the amount of locking is higher than necessary; such a pessimistic locking policy increases the danger of deadlocks. Additionally, a client that holds a lock on a given processor cannot relinquish it temporarily when the lock is needed by one of its suppliers. As a result, certain scenarios, e.g. separate callbacks, cannot be implemented.

This chapter presents two refinements of the locking policy: a type-based mechanism to specify which arguments of a routine call should be locked, and a lock passing mechanism for safe handling of complex synchronisation scenarios with mutual locking of several separate objects. When combined, these two refinements greatly improve the expressive power of SCOOP, give the programmers more control over the computation, and increase the potential for parallelism, thus reducing the risk of deadlock.

7.1 Eliminating unnecessary locks

In this section, we present the first refinement of the locking policy: a mechanism for specifying which formal arguments of a routine should be locked. The selective locking relies on attached types recently introduced in Eiffel[1] [53]; it is fully compatible with other object-oriented concepts such as inheritance, polymorphism, and dynamic binding. The application of attached types to concurrency is a result of joint work with Bertrand Meyer; the basic idea was described in [96, 109]. Here, we refine the mechanism, integrate it with the rest of the SCOOP framework, and study its impact on the advanced language features.

7.1.1 (Too much) locking considered harmful

Recall that SCOOP, as defined so far, requires all formal arguments of a feature to be reserved before the feature is applied (see the feature application rule 6.1.4). This rule is too restrictive. Consider the feature r in Figure 7.1. The handlers of x, y, and z must be locked on behalf of the executing processor before the body of r is executed. (Rule 6.1.5 also requires *some_precondition* to hold before executing r. For the moment we will ignore this requirement; we will come back to it in section 7.1.3.) Is it really necessary to lock all the arguments?

[1] *Spec*♯ has a similar type mechanism: *non-nullable types*.

```
r (x: separate X; y: separate Y; z: separate Z)
    require
        some_precondition
    local
        my_y: separate Y
        my_z: separate Z
    do
        x.f
        my_y := y
        x.g
        my_z := z
        s (z)
    end
```

Figure 7.1: Greedy locking

The body of *r* contains two calls on *x*, therefore *x* needs to be locked. There is no way around it: we must ensure that no other processor is currently using *x*. On the other hand, *y* only appears as source of an assignment; no calls on *y* are made. Similarly, *z* only appears as source of an assignment and as actual argument of a feature call. It seems that only the processor that handles *x* needs to be locked; it is not necessary for *y* and *z* because the body of *r* does not perform any calls on them.

The eager locking policy has two major drawbacks. First of all, it increases the danger of deadlocks: the more resources a routine requires, the more likely it is to end up in a deadlock. Secondly, references cannot be passed around without locking the corresponding objects; programmers have no real control over the locks.

7.1.2 Semantics of attached types

We rely on the attached type mechanism [53] to support selective locking of feature arguments. We require all attached formal arguments to be locked before a feature is applied; no detachable formal arguments are locked. The refined rule 7.1.1 captures precisely the necessary and sufficient conditions for a safe application of features.

Definition 7.1.1 (Feature application rule (refined)) *Before a feature is applied, its attached formal arguments must be reserved by the supplier, and its precondition must hold.*

Together with the call validity rule 6.5.3, the feature application rule 7.1.1 ensures atomicity: a routine represents a critical section with respect to all the processors that handle its attached formal arguments; as a result, all the calls within the routine are guaranteed to be handled atomically. But the new semantics of attached and detachable types implied by these rules is not a mere "hack" to optimise locking. It should be viewed as a generalisation of the sequential semantics; it turns out to be necessary for other purposes as well:

- *A solution of the void separate argument problem.*
  As pointed out in section 5.8.3, it is unclear what the semantics of locking should be in

the case of void arguments. Rule 7.1.1 solves the issue: detachable arguments are ignored by the locking mechanism altogether.

- *The development of a lock passing mechanism.*
  The decision whether to pass or not to pass the locks is based on the argument types (see section 7.2).

- *A sound reasoning technique for separate feature calls.*
  The refined locking policy is essential to ensure the soundness of the sequential-style proof technique for asynchronous calls, developed in section 8.2.

Let us rewrite the example from figure 7.1 to make use of the new mechanism. Figure 7.2 shows an optimised version of *r*. Now, following the rule 7.1.1, the handler of *x* is locked before *r* is executed because *x* is an attached formal argument; the handlers of *y* and *z* are not locked because *y* and *z* are detachable.

```
r (x: separate X; y: ?separate Y; z: ?separate Z)
    local
        my_y: ?separate Y
        my_z: ?separate Z
    do
        x.f
        my_y := y
        x.g
        my_z := z
        s (z)
    end
```

Figure 7.2: Selective locking

The new rule for feature application gives programmers an increased control over locking; it allows a precise specification of resources needed by a routine, and it enables the implementation of interesting scenarios that used to be impossible (or very difficult) to implement. It is now possible to pass around a reference without locking the corresponding object; in SCOOP_97 an inelegant hack was necessary whereby the reference was wrapped in an expanded object; it only worked for separate references due to the restrictions imposed on expanded types by the separateness consistency rule SC4 (4.2.9).

One can also observe the increased potential for parallelism: since the handlers of *y* and *z* are not locked, other clients may use them while *r* is being executed. There is no harmful interference between these clients and the processor executing *r* because the latter never performs any calls on *y* or *z*. Therefore, the increased amount of parallelism does not compromise the safety guarantees.

7.1.3 Support for inheritance and polymorphism

To be usable in practice, the refined locking mechanism must be compatible with inheritance, polymorphism, and dynamic binding. Clients must not be cheated upon in the presence of polymorphic calls, i.e. the safety guarantees should be preserved even if a redefined version of a

feature is applied instead of the original version assumed by the client. The signature conformance rule of Eiffel (rule 8.14.4 /VNCS/ in [53]) allows a covariant redefinition of argument types, provided that the redefined arguments are declared as detachable. The rule for result types is less strict: the redefined type has to conform to the original one but does not need to be marked as detachable. How well do these rules fit into to the enriched SCOOP type system? We will fully address this topic in section 9.1 devoted to polymorphism in SCOOP; here we only consider one component of a type — the detachable tag — and the conformance between attached and detachable types. Recall the subtype relation defined in section 6.11: type $U = (\delta, \beta, Y)$ is a subtype of type $T = (\gamma, \alpha, X)$ if and only if the three components of U — the detachable tag δ, the processor tag β, and the class type Y — conform to the corresponding components of T. The conformance relation on detachable tags says that attached (!) conforms to detachable (?), e.g. $(!, \bullet, X)$ is a subtype of $(?, \bullet, X)$, and $(!, \top, X)$ is a subtype of $(?, \top, X)$.

The validity rule should only permit the redefinition of feature arguments and result types in a way that does not penalise the clients of a given feature. How could a client be penalised? The first possibility is to redefine a detachable formal argument into an attached one. The second possibility is to redefine the result type of a query from attached to detachable. In both cases, as a result of a polymorphic call, an attachment (assignment or argument passing) from a detachable source to an attached target may occur. This violates the type safety; hence, such redefinitions should be prohibited. In a concurrent context, the redefinition of formals from detachable to attached has another unpleasant side effect: the objects represented by the redefined arguments are locked although the signature of the original routine stipulates no locking. This means more waiting than the client expects; it is clearly unacceptable.

On the other hand, a redefinition of a result type from detachable to attached does not cause any problems: the client assumes the result to be detachable, so providing an attached result simply gives a stronger guarantee. Similarly, a redefinition of a formal argument from attached to detachable is safe. The client has to use an attached actual argument as required by the original signature; the redefined feature may choose to expect less and only require a detachable argument. The argument is not locked — although the signature of the original feature suggests it — but this causes no harm for the client. On the contrary: the amount of locking is reduced, so the client needs to wait less than with the original feature. (Strictly speaking, the amount of locking is *not higher* than in the original feature.) To satisfy the call validity rule 6.5.3, the new body must not perform any calls on the redefined arguments. The following rule captures the intended validity constraints for feature redefinition.

Definition 7.1.2 (Feature redefinition rule (tentative)) *The return type of a feature may be redefined in a descendant from detachable to attached. The type of a formal argument may be redefined from attached to detachable.*

Routine *r* in figure 7.2 is a valid redefinition of the original routine from figure 7.1: the original version takes two arguments of type $(!, \top, Y)$ and $(!, \top, Z)$; the redefined version takes arguments of type $(?, \top, Y)$ and $(?, \top, Z)$ respectively. All calls within the redefined body satisfy the rule 6.5.3.

The feature redefinition rule 7.1.2 seems to capture the necessary requirements for a safe redefinition of features. There are, however, two outstanding problems:

- The use of precursor calls is not always possible[2].

[2]We are grateful to Bernd Schoeller for pointing out this problem.

- The inherited precondition and postcondition clauses that involve calls on the redefined arguments may become invalid.

Once again, these problems may arise out of other typing issues but we discuss here only the problem of detachability. (Section 9.1 addresses the remaining topic: the redefinition of processor tags in formal argument and result types.) Consider the feature *r* in figure 7.3 to be a redefined version of *r* from figure 7.1. The redefined version lists a precondition *new_precondition*; this weakens the requirements put on clients: the precondition is understood as *some_precondition* **or else** *new_precondition*. The body of *r* follows a simple pattern: if *new_precondition* holds, some particular actions are taken; otherwise, the original version is called through **Precursor** $(x,\ y,\ z)$. But the precursor call is rejected by the compiler because the types of actual arguments *y* and *z* ($(?, \top, Y)$ and $(?, \top, Z)$) do not conform to the types of the corresponding formals ($(!, \top, Y)$ and $(!, \top, Z)$ respectively) in the original feature. To perform a call to **Precursor**, explicit downcasts (object tests) should be applied to *y* and *z*, as illustrated in figure 7.4 (see also section 6.7 for a detailed discussion of the object test mechanism).

```
r (x: separate X; y: ?separate Y; z: ?separate Z)
    require else
        new_precondition
    do
        if new_precondition then
            -- do something here
        else
            Precursor (x, y, z) -- Invalid!
        end
    end
```

Figure 7.3: Problem with **Precursor** calls

While the problem of invalid precursor calls is easy to detect (it amounts to a simple type-check) and to deal with, the problem of contract inheritance is trickier. Consider again the programming pattern used in figure 7.3. The **else** branch is taken if *some_precondition* holds (because we know that *new_precondition* is false and *some_precondition* **or else** *new_precondition* holds); but this assumption is only valid if *some_precondition* does not involve calls on *y* or *z*. What happens if such calls do appear in *some_precondition*? For example, take *some_precondition* to be *x*.*is_empty* **and** *y*.*is_empty*. What is the meaning of *y*.*is_empty* in the context where *y* becomes detachable? According to the call validity rule 6.5.3, *y*.*is_empty* is valid only if the type of *y* is attached, which obviously is not the case in the redefined version of *r*. Nevertheless, in the context of the original routine where *y* was attached, it was a valid call. It seems that, due to the redefinition of formal arguments from attached to detachable, it is possible to invalidate inherited assertions that involve calls on redefined arguments. There are two alternative ways to prevent it:

1. Assume that all inherited assertions involving calls on detachable formal arguments hold vacuously. For example, if *y* is detachable, *x*.*is_empty* **and** *y*.*is_empty* simply reduces to **True**.

2. Prohibit the redefinition of formal arguments appearing as targets of feature calls in preconditions or postconditions.

```
r (x: separate X; y: ?separate Y; z: ?separate Z)
    require else
        new_precondition
    do
        if new_precondition then
            -- do something here
        elseif {aux_y: separate Y}y and then {aux_z: separate Z}z then
            Precursor (x, aux_y, aux_z)
        end
    end
```

Figure 7.4: Correct use of **Precursor**

The first solution is compatible with the DbC rule for preconditions: inherited preconditions may be weakened. Unfortunately, inherited postconditions may be weakened too, which is clearly against the rules of DbC. The second solution does not suffer from that drawback. Nevertheless, it forces programmers to preserve the attached type of a formal argument even if the redefined version of the routine does not rely on any properties of that argument anymore. It might have no importance in the sequential context but in a concurrent context, where the detachability of an argument implies less locking, such a restriction is burdensome. Essentially, once a formal argument has been used in a precondition or a postcondition, it cannot be redefined from attached to detachable in descendants; there is no possibility to reduce the locking requirements of the routine.

In practice, we may expect that an attached separate formal argument involved in a postcondition will never be redefined into a detachable one, simply because all redefined versions of a routine have to satisfy at least the original postcondition; there is no way to satisfy that postcondition without the guarantee that no other clients change the state of the object represented by the formal argument. Such a guarantee may only be obtained by locking the argument for the duration of the call, which requires the argument to be declared as attached. On the other hand, a routine that does not lock the given formal argument and needs no assumptions about its state, may simply ignore the precondition clauses concerning that argument, i.e. take them to be trivially true. Therefore, the two solutions presented above can be combined to yield a sound and practical redefinition rule 7.1.3.

Definition 7.1.3 (Feature redefinition rule (detachable tags)) *The return type of a feature may be redefined from detachable to attached. The type of a formal argument may be redefined from attached to detachable, provided that no calls on that argument appear in the inherited postcondition.*

This will be part of the update to the general Eiffel feature redefinition rule (see section 9.1.2). The auxiliary rule 7.1.4 clarifies the meaning of inherited precondition clauses.

Definition 7.1.4 (Inherited precondition rule) *Inherited precondition clauses involving calls on a detachable formal argument hold vacuously.*

This rule put a higher burden on the redefiner of a routine: if the inherited precondition involves calls on an argument that has been redefined into detachable, the new routine body may assume less but must establish the same (or stronger) postcondition.

Discussion

In addition to the solution based on attached types, we considered two alternative ways of specifying which formal arguments should be locked. The first option is a compiler optimisation based on the *Business Card principle* of SCOOP_97 (see section 5.8.2): if the body of r does not perform any calls on x, then the processor handling x does not need to be locked before r is executed. This is decided by the compiler; programmers need no additional type annotations. Unfortunately, this solution is not acceptable for at least three reasons:

- Programmers have no control over locking; the locking behaviour depends on the actual version of the routine chosen at run time.
- Without looking at the implementation of the feature, a client cannot see whether a formal argument is locked or not; the interface is not precise enough to infer all the necessary information. This violates the principle of information hiding.
- The client might be deceived: a redefined version of the feature may lock an argument that the original version does not lock.

The second alternative relies on the extensive use of preconditions. To lock the processor handling a formal argument x, an assertion of the form *is_reserved* (x) or — in a more object-oriented style — x. *is_reserved* , must appear in the precondition clause of the enclosing routine.

```
r (x: separate X; y: separate Y; z: separate Z)
    require
        is_reserved (x)
    do
        ...
    end
```

Such assertions force the processor executing r to block until the corresponding formal arguments are reserved on behalf of that processor. This solution is compatible with polymorphism and dynamic binding: removing *is_reserved* (x) from the redefined precondition eliminates the lock requirement on x. Nevertheless, the "locking" part of the redefined precondition should shadow the original one rather than being **or**-ed with it; this obfuscates the resulting precondition. Also, this solution is too verbose; it is much easier to read and write crisp code like

```
s (x, y, z: separate X; a: ?separate A) do ... end
```

than clumsy code like

```
s (x, y, z: separate X; a: separate A)
    require
        is_reserved (x) and is_reserved (y) and is_reserved (z)
    do
        ...
    end
```

Attached types provide a sound solution which also integrates best with other object-oriented mechanisms.

7.2 Lock passing

The next step to refine the locking policy is to let clients relinquish their locks temporarily and pass them on to a supplier. The mechanism presented here relies on the selective locking introduced in section 7.1: clients use attached and detachable types to decide whether a lock passing should take place. The proposed mechanism reduces the danger of deadlocks and allows the implementation of interesting synchronisation scenarios, e.g. separate callbacks, without compromising the atomicity guarantees. We generalise the semantics of argument passing in a way that accomodates the lock passing mechanism and ensures the soundness of the proof technique for asynchronous calls developed in section 8.2.

The need for lock passing was initially identified by Phil Brooke and later reflected in his CSP semantics of SCOOP_97 [34] in the form of transitive locking, whereby suppliers can "snatch" a lock from their clients when necessary. Although we use the same name for our mechanism, we follow a different approach here; the differences between the two solutions are discussed in section 7.3.

7.2.1 Need for lock passing

In SCOOP_97, a routine holds exclusive locks on its separate suppliers (which have to be formal arguments of the routine) during the execution of its body. As pointed out in section 4.2.3, this policy ensures that no other client can jump in and modify the state of a supplier object. While this *atomicity* guarantee is convenient for reasoning about concurrent software — we may apply similar techniques as for sequential programs — it unnecessarily limits the expressiveness of the model and increases the danger of deadlocks. Figure 7.5 illustrates a typical problem caused by cross-client locking. Calls to $x.f$, $x.g$, and $y.f$ are asynchronous (f and g are commands),

```
r (x: separate X; y: separate Y)
    do
        x.f
        x.g (y)     -- x waits for y to become available.
        y.f
        ...
        z := x.some_query    -- Current waits for some_query to finish.
                             -- DEADLOCK!
    end
```

Figure 7.5: Deadlock caused by cross-client locking

so the client does not wait for their completion. Following the *wait by necessity* principle, the client only waits for the result of the query call $x.some\_query$. Unfortunately, this causes a deadlock because x's handler is not able to evaluate *some_query* before finishing all the previously requested calls on x; one of these calls, $x.g\ (y)$, needs a lock on y's handler, currently

held by the client. But the client cannot unlock y before finishing r's body. So, the client is waiting for x's handler and vice-versa; none of them will ever make any progress.

In fact, getting into a deadlock situation is even simpler; no cross-client locking is necessary. It suffices to pass **Current** as actual argument of a separate query call, as illustrated in figure 7.6. Since feature g called on x needs to lock the processor that handles **Current**, it will block until that processor can be reserved. But it will never be the case because the client is waiting for the completion of g; hence the client's handler is not idle and cannot be reserved. Again, we have a deadlock; this time, it is caused by a callback (or rather a "lock-back") of g's handler on **Current**'s handler. The body of g does not even need to perform any real callback in order to cause a deadlock!

```
s (x: separate X)
    do
        z := x.g (Current)  -- x waits for Current.
                            -- Current waits for x to finish. DEADLOCK!
    end
```

Figure 7.6: Deadlock caused by a callback

Meyer [94] suggested to solve the callback problem by applying the *Business Card principle* (see section 4.2.5) which stipulates that clients may only pass a reference to **Current** to features that do not lock the corresponding formal argument, i.e. whose body does not contain any calls on that argument. Unfortunately, the principle does not work well with inheritance and polymorphism (see section 5.8.2). Furthermore, it actually prohibits separate callbacks rather than accomodating them.

7.2.2 Mechanism

In the two examples above, a deadlock occurs at the moment when the client waits for one of its suppliers. Since the client is waiting, it does not perform any operations. Therefore, it makes no use of the locks it holds. If the client could temporarily pass the lock on y (in Figure 7.5) respectively on **Current** (in Figure 7.6) to its supplier x, the supplier would be able to execute the requested feature, return the result, and let the client continue, thus avoiding the deadlock. This basic idea is simple but, besides solving the problem of cross-client locking and separate callbacks, a lock passing mechanism has to satisfy further requirements:

- *It must not compromise the atomicity guarantees.*
  A sound reasoning about feature calls is only possible if other clients do not interfere, i.e. the accesses to a given object are atomic. This immediately rules out a solution whereby a client passes a lock on an object to a supplier and then continues its own execution: it would be impossible to decide statically how the client's and the supplier's calls on the locked object are ordered. As a result, neither the client nor the supplier would be able to ensure the correctness of their calls. Additionally, assertions involving calls on the concerned object would not be usable.

- *Clients must be able to decide to pass or not to pass a lock.*
  It must be clear from the program text whether a lock passing occurs. The mechanism

should be controlled by clients, i.e. they should have the choice to pass or not to pass a lock to a supplier that needs it. Letting a supplier snatch a lock without asking for the client's permission is unacceptable; it complicates the reasoning about programs and it may lead to an arbitrary interleaving of accesses to the locked object, thus compromising the atomicity.

- *The mechanism should increase the expressiveness of the language, not restrict it.* Lock passing should enable the implementation of additional interesting synchronisation scenarios not supported in the basic model. On the other hand, all scenarios implementable in the basic model should be expressible in the extended framework as well.

- *The solution must be simple and well integrated with other language mechanisms.* The solution must be sound in the presence of polymorphism and dynamic binding; it has to be compatible with the rules of DbC as well.

We propose the following solution: if a feature call $x.f\ (a_1, ..., a_n)$ occurs in the context of the routine r where some actual argument a_i is *controlled*, i.e. a_i is attached and locked by r (see definition 6.5.2), and the corresponding formal argument of f is declared as attached, the client's handler (the processor executing r) passes all currently held locks (including the implicit lock on itself) to the handler of x, and waits until f has terminated. When the execution of f is complete, the client's handler resumes the computation.

Let's see how our mechanism solves the problems of cross-client locking and separate callbacks. Feature r in figure 7.7 is identical with feature r from figure 7.5 but it does not deadlock, due to lock passing: the call $x.g\ (y)$ is executed synchronously, with the client passing on all its locks to x for the duration of g. No deadlock occurs when the client evaluates $x.some\_query$ because the handler of x is not blocked anymore; the execution of $x.g\ (y)$ has terminated so $x.some\_query$ can be applied. Similarly, the routine s in Figure 7.8 does not deadlock any-

```
-- in class C
r (x: separate X; y: separate Y)
    do
        x.f
        x.g (y)        -- Current passes its locks to x
                       -- and waits until h terminates .
        y.f
        ...
        z := x.some_query      -- No deadlock here
    end

-- in class X
g (y: separate Y)
    do
        ...
    end
```

Figure 7.7: Cross-client locking without deadlock

more because $x.h$ (**Current**) results in the lock passing which lets x's handler obtain a lock on

Current without waiting. (In this particular case, the client and the actual argument are both handled by the same processor; we assume that every processor, when non-idle, implicitly holds a lock on itself.)

That last example raises an interesting issue: if the body of *h* indeed performs a callback, i.e. a call on a target handled by the processor that has passed its locks, how should such a call be treated? Consider the call *c*.*f* (...) in figure 7.8; does it have the usual asynchronous semantics whereby a request to execute *f* is queued on *c*'s handler? If yes, then the problem of deadlock is not really solved but just postponed. To avoid this, *c*.*f* (...) should be performed synchronously, i.e. scheduled it for an immediate execution, so that *c*'s handler — the one that has initially passed its locks and is waiting for the termination of *x*.*h* (**Current**) — has a chance to execute it. So, the call is separate but synchronous; this may seem a bit disturbing. A closer inspection, however, reveals the underlying reason for applying this semantics: the target is handled by a processor that holds a lock on the the current processor. This is just like for non-separate calls, where the target's handler — which happens to be the current processor itself — holds a lock on the current processor. Therefore, we can generalise the applicability of the synchronous call semantics to all the calls whose target's handler holds a lock on the current processor. (All such calls may be viewed as separate callbacks.) Note that this rule permits a nested (or even recursive) lock passing, i.e. any feature call within the body of *h*, including *c*.*f* (...) , may result in a lock passing whereby the locks obtained at the previous step are passed further. No limit on the depth of lock passing is imposed; the atomicity guarantees are preserved because the client always blocks.

```
-- in class C
s (x: separate X)
    do
        z := x.h (Current)  -- x gets lock on Current.
                            -- No deadlock here
    end

-- in class X
h (c: separate C): Z
    do
        c.f (...)
        ...
    end
```

Figure 7.8: Callback without deadlock

One more point needs to be clarified: whenever the lock passing occurs, the client passes *all* its locks to the supplier, not only the locks corresponding to the particular arguments that triggered the mechanism. Such a generous behaviour of clients eliminates more potential deadlocks than passing just the specified locks. The client does not use any locks anyway while it is blocked so it does not hurt to pass locks "just in case"; on the contrary, the supplier might make use of these additional locks in the body of the requested routine that perhaps would deadlock otherwise.

Definition 7.2.1 captures the semantics of feature calls, reflecting the refined meaning of argument passing and the additional synchrony requirement.

Definition 7.2.1 (Feature call semantics (refined)) *A feature call $x.f\ (\overline{a})$ results in the following sequence of actions performed by the client's handler P_c:*

1. *Argument passing: bind the formal arguments of f to the corresponding actual arguments $\overline{a}$. If any attached formal argument corresponds to a controlled actual argument of a reference type, pass all the currently held locks (including a lock on P_c) to the supplier's handler P_x.*
2. *Feature request: ask P_x to apply f to x.*
   (a) *Schedule f for an immediate execution by P_x and wait until it terminates, if any of the following conditions holds:*
      - *The call is non-separate, i.e. $P_c = P_x$.*
      - *The call is a separate callback, i.e. P_x already held a lock on P_c at the moment of the call.*
   (b) *Otherwise, schedule f to execute after the previous calls on P_x.*
3. *Wait by necessity: if f is a query, wait for its result.*
4. *Lock revocation: if lock passing occurred in step 1, wait for f to terminate, then revoke the locks from P_x.*

7.2.3 Lock passing in practice

Figure 7.9 illustrates the possible combinations of separate and non-separate calls with and without wait by necessity and lock passing. The current object (an instance of *C*) is handled by the processor P_c, and *x*, *y*, *my_c*, and *my_z* are handled by (different) processors P_x, P_y, P_{my_c}, and P_{my_z} respectively. The calls appearing in the body of *r* have the following semantics:

- (Command call *my_x.f* (5)) The call is non-separate because *my_x* is handled by P_c; the current execution state is saved on P_c's call stack, and *my_x.f* (5) is executed synchronously. No lock passing occurs because the actual argument is expanded. The request queues are not involved in this operation, hence they remain empty:
  $P_c: \ -$ $\quad$ $P_x: \ -$
- (Command call *my_x.g* (*x*)) Similar to the previous call but lock passing occurs. However, it is vacuous because both the client and the supplier objects are handled by P_c. The request queues are not involved in this operation, hence
  $P_c: \ -$ $\quad$ $P_x: \ -$
- (Query call *my_x.h* (**Current**)) Similar to the previous call but wait by necessity applies. Lock passing occurs but is vacuous. The call is non-separate, therefore the request queues are not involved.
  $P_c: \ -$ $\quad$ $P_x: \ -$

```
-- in class C
my_x: X
my_z: separate Z
my_c: separate C
i: INTEGER

r (x: separate X; y: separate Y)
    do
        my_x.f (5)          -- non-separate, no wait by necessity, no lock passing
        my_x.g (x)          -- non-separate, no wait by necessity,
                            -- lock passing (vacuous)
        i := my_x.h (Current)   -- non-separate, wait by necessity,
                                -- lock passing (vacuous)

        x.f (10)            -- separate, no wait by necessity, no lock passing
        x.g (my_z)          -- separate, no wait by necessity, no lock passing
        x.g (y)             -- separate, no wait by necessity, lock passing
        x.m (y)             -- separate, no wait by necessity, no lock passing
        i := x.h (my_c)     -- separate, wait by necessity, no lock passing
        i := x.h (Current)      -- separate, wait by necessity, lock passing
    end

-- in class X
f (i: INTEGER) do ... end

g (a: separate ANY) do ... end

h (c: separate C): Z
    do
        c.f (...)
        ...
    end

m (a: ?separate ANY) do ... end
```

Figure 7.9: Lock passing example

- (Command call $x.f\ (10)$) The call is separate because $P_c \neq P_x$. No lock passing occurs because the actual argument is expanded.
  $P_c: -\quad P_x: [x.f(10)]$

- (Command call $x.g\ (my\_z)$) Similar to the previous call. No lock passing occurs because the actual argument my_z is not controlled.
  $P_c: -\quad P_x: [x.f(10)][x.g(my\_z)]$

- (Command call $x.g\ (y)$) Similar to the previous call but lock passing occurs because the actual argument y is controlled and the corresponding formal is attached. P_c cannot move

to the next operation until P_x has terminated the application of $x.g\ (y)$ (after servicing all the previous calls in its request queue).

P_c : – P_x : $[x.f(10)][x.g(my\_z)][x.g(y)]$ (after the *feature request* step)
P_c : – P_x : – (after the *lock revocation* step)

- (Command call $x.m\ (y)$) No lock passing occurs because the formal argument a is detachable.
  P_c : – P_x : $[x.m(y)]$
- (Query call $x.h\ (my\_c)$) Wait by necessity applies. No lock passing occurs because the actual argument *my_c* is not controlled.
  P_c : – P_x : $[x.m(y)][x.h(my\_c)]$ (after the *feature request* step)
  P_c : – P_x : – (after the *wait by necessity* step)
- (Query call $x.h\ (\textbf{Current})$) Similar to the previous call but lock passing occurs because the actual argument **Current** is controlled.
  P_c : – P_x : $[x.h(Current)]$ (after the *feature request* step)
  P_c : – P_x : – (after the *lock revocation* step)

That last call is particularly interesting because it involves a separate callback: during the application of $x.h\ (\textbf{Current})$, P_x performs the call $c.f\ (...)$ where c corresponds to the actual argument of the call $x.h\ (\textbf{Current})$. The target of the call $c.f\ (...)$ is handled by P_c, and P_c holds a lock on the client's handler P_x (note the inversed roles of P_c and P_x). Therefore, the call is handled as a separate callback (according to step 2a in the definition 7.2.1), i.e. P_c immediately executes the feature requested by P_x; the latter waits until the feature has terminated. It is important to notice that the request queue of P_c is not involved in handling this request; the feature is handled using the call stack, just like a non-separate call. That is why P_c's queue remains empty throughout the execution of $x.h\ (\textbf{Current})$.

Figure 7.10 recapitulates the possible type combinations of formal and actual arguments, and the resulting semantics of argument passing (*yes* stands for "lock passing takes place", *no* stands for "no lock passing").

	formal attached	formal detachable
actual (reference type) controlled	**yes**	*no*
actual (reference type) uncontrolled	*no*	*no*
actual expanded	*no*	*no*

Figure 7.10: Lock passing combinations

The lock passing mechanism influences the semantics of feature calls so that certain calls, e.g. $x.g\ (y)$ in figure 7.9, have a different meaning in SCOOP than in SCOOP_97. Nevertheless, it is possible to emulate the original semantics — at a cost of a few additional lines of code — as illustrated in figure 7.11. The original feature *g* from figure 7.9 has been replaced by a pair of features: *g* and *blocking_g*. The formal argument of *g* is now detachable, so the call $x.g\ (y)$ does not involve lock passing, even though the actual argument *y* is controlled. The call to the auxiliary feature *blocking_g* will later lock *y* but it does not influence the semantics of $x.g\ (y)$ as seen by the client; the call $x.g\ (y)$ is non-blocking, just as it would be in SCOOP_97. Note the use of an object test in the body of *g* for downcasting a detachable type to an attached type.

```
-- in class C
r (x: separate X; y: separate Y)
    do
        ...
        x.g (y)        -- No lock passing here.
        ...
    end

-- in class X
g (y: ?separate Y)
    local
        aux_y: separate Y
    do
        if {aux_y: separate Y} y then
            blocking_g (aux_y)
        end
    end

blocking_g (y: separate Y)
    do
        ...  -- body of original g
    end
```

Figure 7.11: Emulating SCOOP_97 semantics

Discussion

The proposed mechanism fulfils all the additional requirements discussed at the beginning of this section. First, it does not compromise the atomicity because locks are passed to the client's handler only for the duration of f; since the client is blocked in the meantime, there is no danger of harmful interleaving with other clients. Of course, the supplier is free to perform any sequence of calls on the locked objects but the client knows that all these calls will be executed before its own subsequent calls; additionally, the postcondition of f stipulates what the supplier may or may not do with these objects. The reasoning about calls that involve lock passing is therefore similar to reasoning about sequential calls (see section 8.2). Second, it is clear from the program text whether a lock passing occurs: the controllability of actual arguments is immediately deducible from their type; the type of the formal arguments of f is known from f's signature. Therefore, a client can decide to pass or not to pass its locks simply by using controlled or uncontrolled actual arguments; alternatively, it may use a feature that takes detachable formals. Note the absence of lock passing for actual arguments of an expanded type (even though they are always controlled). This reflects the copy semantics of such arguments: the corresponding formals are bound to copies of actuals. Since these copies are non-separate from the supplier, no lock passing is necessary to give the supplier the control over them.

Lock passing increases the expressiveness of the programming framework: several scenarios not supported by SCOOP_97 — including the cross-client locking and separate callbacks

illustrated in figures 7.7 and 7.8 — are now implementable. Section 7.2.3 demonstrates that the locking policy of SCOOP_97 can be simply emulated in SCOOP; therefore, the backward compatibility is preserved and the mechanism does not limit the expressive power of our framework. Finally, not only does the lock passing mechanism combine well with polymorphism and dynamic binding (thanks to the reliance on the attached type semantics), it is instrumental in providing a sound basis for the proof technique for concurrent software, described in section 8.2.

7.3 Related work

This chapter is based on our previous work on the locking policy for SCOOP, described in an earlier article [109] and a technical report [110] which discussed the use of detachable types in SCOOP but did not cover the problems related to inheritance and polymorphism; the lock passing mechanism was only described shortly, without considering the complex scenarios discussed here. The report also presented a basic mechanism for shared locking, based on a refined notion of a *pure query* and a new semantics for the **only** clauses (used in the sequential Eiffel to express the frame properties of features). The shared locking mechanism proved unsound in the presence of polymorphism, therefore we do not consider it here. We are currently working on its refinement that provides a full support for inheritance and polymorphism.

Meyer [96] discusses the attached type mechanism, in particular its use for eliminating *catcalls*. He also describes the results of our earlier discussions about the application of attached types to concurrency. The problem of contract redefinition in a concurrent context is not discussed but the article prompted us to have a closer look at the contract inheritance mechanism. Meyer's rule for argument redefinition requires a covariantly redefined type of a formal argument to be marked as detachable. (The Eiffel standard has adopted a corresponding signature conformance rule 8.14.4 /VNCS/.) Inherited assertions involving calls on detachable arguments are evaluated using an implicit object test. For example, for attached x and detachable y, the expression

$x.\,is\_empty$ **and** $y.\,is\_empty$

is understood as

$x.\,is\_empty$ **and** $(\{aux\_y\colon\ Y\}y$ **implies** $aux\_y.\,is\_empty)$

hence $x.\,is\_empty$ if y is void. Besides being complicated, this solution is inconsistent with DbC: as demonstrated in section 7.1.3, it may lead to postcondition weakening. Our rules 7.1.3 and 7.1.4 may be combined with the signature conformance rule 8.14.4 /VNCS/ to ensure the consistency with DbC and to simplify the treatment of inherited contracts (section 9.1 discusses this topic in detail). Our technique, initially developed to solve a concurrency issue, proves useful in a sequential context as well.

Brooke et al. [34] propose a different lock passing mechanism as part of their CSP semantics for SCOOP. The authors apply a transitive locking by default, i.e. if a client object c holds locks on the supplier objects x and y, and x requests a lock on y, then x "snatches" that lock from the client object. An advantage with respect to our approach is the increased potential parallelism: the asynchronous semantics still applies to calls that involve lock passing, whereas our solution forces such calls to be synchronous. Nevertheless, programmers have no control

over the mechanism: lock passing happens whenever possible, even if there is no need for it. Furthermore, the calls on y issued by c and x may be arbitrarily interleaved; even though c temporarily loses its lock on y, it is impossible to predict when exactly it happens.

```
-- in class C
r (x: separate X; y: separate Y)
        -- Assume that y = x.my_y, so both Current and x
        -- use the same separate object.
    do
        x.f             -- Body of f requests a lock on y.
                        -- Lock passing may occur here
        y.f
                        -- or here
        y.g
                        -- or here
        y.h
                        -- or here
    end

-- in class X
my_y: separate Y

f
    do
        ...
        s (my_y)        -- Snatch the lock from the client.
        ...
    end

s (y: separate Y) do ... end
```

Figure 7.12: Problems with transitive locking

Figure 7.12 illustrates the problem: if c executes several calls on y after the call $x.f$, the lock passing may occur either before the call to $y.f$, before the call to $y.g$, before the call to $y.h$, or after that. In fact, it may even not happen at all; if the execution of f by x's handler is very slow then the client's handler may be able to schedule all its calls on y and terminate the body of r before x attempts to snatch the lock. As a result, one cannot reason about the order of separate calls on y; the assertional reasoning about concurrent code becomes intractable. The problem is particularly acute in the context of inheritance and polymorphism: even if the original version of f in class X does not perform any calls that imply lock passing, a redefined version in a descendant may do so. The clients of the original class are unaware of that and do not expect any lock passing. Our solution avoids such problems: the clients make explicit decisions about lock passing; a redefined version of a routine cannot require more locks than the original version. Furthermore, Brooke et al. only support passing locks on separate objects; the lock on the current processor cannot be passed. As a result, the separate callback scenario depicted in figure 7.6 still leads to a deadlock. Another difference with respect to our solution

is the fact that only one lock is passed at a time but the subsequent calls can demand additional locks.

Discussion

The new semantics of attached types supports precise specification of locking requirements of routines, thus eliminating the unnecessary locking often exhibited in SCOOP_97 programs. The lock passing mechanism lets clients pass on their locks to their suppliers for the duration of a single call. Both mechanisms greatly improve the flexibility of the model by allowing an efficient implementation of many synchronisation scenarios that were difficult (or impossible) to express before. They also preserve the atomicity guarantees but reduce the danger of deadlocks by minimising the amount of locking.

Both the *scoop2scoopli* preprocessor and the *SCOOPLI* library (see chapter 11) support the mechanisms introduced here. A run-time deadlock detection facility which supports lock passing has been devised and integrated with *SCOOPLI* by Daniel Moser [100].

8

Contracts and concurrency

DESIGN by Contract permits enriching class interfaces with assertions which express the mutual obligations of clients and suppliers. Routine preconditions specify the obligations on the routine client and the guarantee given to the routine supplier; conversely, routine postconditions express the obligation on the supplier and the guarantee given to the client. Class invariants express the correctness criteria of a given class; an instance of a class is consistent if and only if its invariant holds in every observable state. The modular design fostered by DbC reduces the complexity of software development: correctness considerations can be confined to the boundaries of components (classes) which can be proved and tested separately. Clients can rely on the interface of a supplier without the need to know its implementation details. We define the correctness of a routine as follows.

Definition 8.0.1 (Local correctness) *Routine r of class C is locally correct if and only if, after the execution of r's body, both the class invariant Inv_C and the postcondition $Post_r$ of that routine hold, provided that both the invariant and the precondition Pre_r were fulfilled at the time of the invocation.*

The sequential proof technique discussed in section 4.2.6 follows this definition; a proof rule for feature calls may be derived:

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{Pre_r[\overline{a}/\overline{f}]\}\ \ x.r(\overline{a})\ \ \{Post_r[\overline{a}/\overline{f}]\}}$$

The above rule says that if a feature *r* is locally correct (according to definition 8.0.1) then a call to *r* in a state that satisfies its precondition will terminate in a state that satisfies its postcondition. Clients simply need to ensure that the precondition holds before the call; they may assume the postcondition in return. It is tempting to apply the same rule to reasoning about concurrent programs. Unfortunately, the standard correctness semantics of assertions breaks down in a concurrent setting (see section 4.2.3). The wait semantics used for separate preconditions in SCOOP_97 palliates the problem but it does not solve it completely: the clash between wait conditions and correctness conditions is a source of inconsistencies; the blocking semantics of postconditions and other assertions reduces the amount of concurrency and may lead to deadlocks (see section 5.6). Also, the restricted proof rule 4.2.1 is too weak to prove interesting properties of calls; it ignores the separate assertions although most properties of interest in a concurrent context are expressed as separate assertions (see section 5.7).

To solve these problems and unleash the full power of contracts, we need to generalise the principles of DbC so that assertions capture the full contract between a client and a supplier, including the required synchronisation. Two things are necessary:

- *A clear semantics of contracts*
  Each kind of assertion should have a uniform semantics applicable in both concurrent and sequential contexts.

- *A modular proof technique*
  The sequential proof rule has to be "lifted" to the concurrent context. The rule must be simple but strong enough to prove interesting properties of programs.

We put an emphasis on the generality of the proposed semantics of contracts. It has to apply in all contexts; furthermore, it should gracefully reduce to the standard correctness semantics when no concurrency is involved. Similarly, the proof rule for sequential programs should be a straightforward refinement of the general rule. We take this requirement one step further: since a human brain is good at sequential reasoning about small portions of code but it cannot deal with a complex combination of parallel processes, the general proof rule should itself have a strong sequential flavour.

8.1 Generalised semantics of contracts

Starting from the sequential semantics of contracts [94] and the critique of contracts in SCOOP_97 (see section 5.6), we analyse the impact of concurrency on each type of assertion and propose a new semantics for it. We demonstrate the influence of the new semantics on other techniques and mechanisms — locking, wait by necessity, polymorphism — and show how it reduces to the traditional semantics in a sequential context. We also discuss the practical implications of our solution.

8.1.1 Preconditions

In a sequential context, a precondition is a correctness condition: the client has to guarantee that the precondition holds at the moment of the call. The supplier expects the precondition to hold when the execution of the routine starts. Since all calls are synchronous, the feature application follows immediately the feature call; no other event may happen in between. Therefore, any property that holds at the call time also holds at the application time; conversely, any property that does not hold at the call time does not hold at the application time. Hence the usual understanding of the client's obligation: the precondition must hold at the call time.

In a concurrent context, the feature call and the feature application do not usually coincide; other events may happen in between. Therefore, a supplier cannot assume that a property satisfied at the call time still holds at the execution time. (This is the source of the *separate precondition paradox* described in section 4.2.3.) Conversely, a property that does not hold at the time of the call may become true later on; the supplier could safely execute the feature at that time. This suggests that preconditions be viewed as a synchronisation mechanism: a called feature cannot be executed unless the precondition holds; a violated precondition delays the execution of the feature. This wait semantics is reflected by the feature application mechanism (definition 6.1.4) and the feature application rule 6.1.5.

SCOOP_97 uses both semantics, depending on whether a given precondition clause involves separate calls. The precondition clause $i > 0$ in figure 8.1 is a correctness condition because

it involves no separate calls, whereas **not** *buffer* . *is_full* has the wait semantics. This distinction is somewhat artificial because *buffer*, although declared as **separate**, may denote a non-separate object at run time, e.g. when the client calls *store* (*ns_buffer* , 10). Should the wait semantics apply here? If yes, then the client will be blocked forever if *ns_buffer* is full. One could argue that this can be avoided because the static type of the actual argument is non-separate; but the same problem may occur even if the actual argument has a separate type, e.g. in the subsequent call *store* (*my_buffer*, 79). Clearly, the wait semantics should be replaced by the correctness semantics here.

```
store ( buffer : separate BUFFER [INTEGER]; i: INTEGER)
        -- Store i in buffer .
    require
        not buffer . is_full
        i > 0
    do
        buffer .put (i)
    end

my_buffer : separate BUFFER [INTEGER]
ns_buffer : BUFFER [INTEGER]
...
store (my_buffer, 24)
store ( ns_buffer , 10)
my_buffer := ns_buffer
store (my_buffer, 79)
```

Figure 8.1: Preconditions

Sometimes, the opposite is required: a correctness condition should be turned into a wait condition. This happens in the presence of polymorphism and feature redefinition. When redefining a feature, the type of a formal argument may be redefined from non-separate to separate; such redefinitions are valid because clients of the original class can only use a non-separate actual argument (see section 9.1.2). What happens to the precondition clauses that involve the redefined formal argument? They are correctness conditions in the original feature; they become wait conditions in the redefined version.

This necessity for wait conditions to be considered as correctness conditions and vice-versa suggests that a uniform semantics should be applied to all preconditions. We propose to adopt the semantics captured by the definition 8.1.1.

Definition 8.1.1 (Semantics of preconditions) *A precondition expresses the necessary requirements for a correct application of the feature. The execution of the feature's body is delayed until the precondition is satisfied.*

The guarantee given to the supplier is exactly the same as with the traditional semantics: the precondition holds at the entry to the feature's body. However, the definition does not force the clients to ensure the precondition before the call. Naturally, a client performing a feature call has certain obligation but the responsibility for establishing the precondition must be shared

```
-- in class X
ns_buffer : BUFFER [INTEGER]
store ( buffer : separate BUFFER [INTEGER]; i: INTEGER)
        -- Store i in buffer .
    require
        buffer_not_full : not buffer . is_full
        i_positive : i > 0
    do
        buffer .put (i)
    end

-- in class C
my_buffer: separate BUFFER [INTEGER]
ns_buffer : BUFFER [INTEGER]
r (x: separate X; buf: separate BUFFER [INTEGER])
    do
        x. store (my_buffer , 10)      -- buffer_not_full  uncontrolled
        x. store ( ns_buffer , 24)     -- buffer_not_full  controlled
        x. store (buf, 79)             -- buffer_not_full  controlled
        x. store (x. ns_buffer , 19)   -- buffer_not_full  controlled
    end
```

Figure 8.2: Controlled clauses

between the client and the environment; the client should be blamed for any contract breaches that it could have avoided but not for anything else. Meyer [94] suggests that only the non-separate precondition clauses bind the client. We go one step further: since the client has a full control over all the *controlled* entities (see definition 6.5.2) — because no other clients may access the corresponding objects in the meantime — it should be blamed for breaches in precondition clauses concerning these entities. Therefore, we extend the client's responsibilities onto all *controlled* clauses.

Definition 8.1.2 (Controlled clause) *For a client performing the call $x.f(\overline{a})$ in the context of a routine r, a precondition clause or a postcondition clause of f is* controlled *if and only if, after the substitution of the actual arguments $\overline{a}$ for the formal arguments, it only involves calls on entities that are controlled in the context of r; otherwise, it is* uncontrolled.

We have chosen clauses as units of controllability, although finer-grained specification could be used instead. Clauses are convenient because they are clearly demarcated in the program text, thus avoiding the risk of confusion. Also, the representation of inherited and immediate assertions in redefined features follows this style; as a result, reasoning in terms of clauses is easier.

Figure 8.2 illustrates the difference between controlled and uncontrolled precondition clauses. For the client executing the call *x. store* (*my_buffer*, 10) in the body of *r*, the first precondition clause of *store* is uncontrolled because, after the substitution of actuals for formals, it contains the call *my_buffer. is_full* whose target *my_buffer* is not controlled in the

context of r. On the other hand, the same precondition clause is controlled in the three remaining calls to *store* because the targets of the involved call to *is_full* are controlled in r: *ns_buffer* is non-separate; *buf* is separate but locked by r; x. *ns_buffer* is separate, but it is non-separate from x and x is controlled, hence x. *ns_buffer* is controlled too (see rule 6.5.2). The second precondition clause $i > 0$ is controlled in all the cases because the expected actual argument must be handled by the same processor as the target x and x is certainly controlled (otherwise, no calls on x would be allowed in r). In this particular case, the actual argument is expanded, hence controlled in any context (see rules 6.10.1 and 6.5.2).

Coming back to the client's responsibilities: each controlled precondition clause must now be ensured by the client. But the calls on x in figure 8.2 are asynchronous; how can a client ensure some property if the previously scheduled calls are still being executed and may change the state of the involved objects? Indeed, the state of x and *buf* at the moment of the call x. *store* (*buf*, 79) may be different from its state at the moment of the feature application; however, the client knows that all its calls on x and *buf* are performed in the FIFO order, therefore sequential reasoning may be used. That is, the state of x and *buf* at the moment of the application of *store* (*buf*, 79) is exactly the same as it was when *store* (*ns_buffer*, 24) terminated. The client only needs to ensure that the properties established by x. *store* (*ns_buffer*, 24) imply the precondition of x. *store* (*buf*, 79). It does not matter that the precondition may not hold at the moment of the call; it will hold when required, i.e. at the moment of the feature application.

The establishment of uncontrolled precondition clauses depends on the emergent behaviour of the whole system. The client cannot take responsibility for such preconditions; calling a feature with an uncontrolled precondition clause may result in indefinite waiting.

Although we use the same name for the proposed *wait semantics* of preconditions, there are three major differences with respect to SCOOP_97:

- The semantics applies to all preconditions; there is no distinction between separate and non-separate clauses.
- Waiting happens on the supplier side. For example, the client executing the call x. *store* (*my_buffer*, 10) in figure 8.2 is not blocked; it is x's handler that waits until the precondition is satisfied.
- Clients are responsible for establishing all the controlled precondition clauses, not only the non-separate ones.

Even though every precondition violation results in waiting, the run-time checking of controlled clauses is optimised to avoid infinite waiting and to reflect the contractual character of such assertions. If a controlled precondition clause is violated when it is supposed to hold, i.e. at the moment of the feature application, waiting is useless because the state of the involved objects cannot change in the meantime; an exception is raised instead. This is precisely how the wait semantics boils down to the traditional sequential semantics when no concurrency is involved: according to definition 8.1.2, all precondition clauses involving only calls on non-separate targets are controlled; when they are violated, an exception is raised. The call *store* (*ns_buffer*, 10) in figure 8.1 can be handled as expected: if the precondition **not** *buffer*. *is_full* is violated, an exception is raised. Similarly, the application of *store* with a negative second argument results in an exception because the precondition clause $i > 0$ is violated. Since the feature application happens immediately after the corresponding feature call, it looks

as if the exception is raised at the moment of the call, just like with the traditional sequential semantics.

8.1.2 Postconditions

The original design of SCOOP_97 applies the standard correctness semantics to separate postconditions; they are evaluated synchronously due to wait by necessity. Meyer [94] also mentions the *separate postcondition paradox*: on return from a separate call, the client cannot be sure that the postcondition clauses still hold, even though they are guaranteed to hold when the call terminates. This is because the processor handling the involved supplier may become free in the meantime and other clients may have jumped in and modified the state of the supplier, thus invalidating the postcondition. (The same problem is identified by Rodriguez et al. [128] as *external interference*; see section 3.2.)

The treatment of postconditions in SCOOP_97 is unsatisfactory for two reasons:

- Separate postconditions are excluded from the proof rule for feature calls (see section 8.2).
- The synchronous evaluation of separate postconditions limits the parallelism and may lead to deadlocks.

The client executing the call *spawn_two_activities* (*york*, *tokyo*) in figure 8.3 wants to launch jobs at two locations and have the guarantee that the jobs will be done. Such guarantees are most naturally expressed as postconditions of *spawn_two_activities* . However, the synchronous semantics of postconditions blocks the client; *do_local_stuff* cannot be called until both clauses *post_1* and *post_2* have been evaluated. The amount of exhibited parallelism is much lower than without the postconditions; in the latter case, the client would be able to move on immediately (but there would be no guarantee that the jobs are done). We propose to evaluate the postconditions *asynchronously*, i.e. treat them similarly to command calls; wait by necessity does not apply. The client can now move on with its own activity (*do_local_stuff*) without waiting for the evaluation of *post_1* and *post_2*; at the same time, it gets the guarantee that both clauses will be satisfied *eventually*. More precisely, they will be satisfied when the execution of the features called within the body of *spawn_two_activities* terminates. Because *york* and *tokyo* are separate, the calls to *do_job* are asynchronous, so *post_1* and *post_2* do not necessarily hold when the client calls *do_local_stuff* ; but this does not matter because *york* and *tokyo* are not used in that feature. The clause *post_1* becomes relevant only at the moment of the call to *get_result* (*york*); the feature cannot be applied until all the previous calls on *york* have terminated and the precondition holds. Therefore, *get_result* cannot be executed until *york* terminates the job requested by *spawn_two_activities* ; when this happens, the postcondition *post_1* holds, i.e. *york*. *is_ready* , is true. Assuming that this property implies the precondition of *get_result* , the latter may be executed. The client is not penalised by the asynchronous evaluation of postconditions; it gets the same guarantees as with the synchronous semantics but "projected" into the future. A postcondition may be assumed to hold immediately after the execution of a feature's body although it may not hold at that point; it will hold when it becomes relevant. This semantics can be refined to allow the individual evaluation of postcondition clauses. In the above example, the state of *tokyo* is irrelevant when *get_result* (*york*) is executed (it will only become important later on for *get_result* (*tokyo*)). For the moment, the

client is only interested in the first postcondition clause; if it holds, *get_result* (*york*) should proceed without waiting for *tokyo*. This enables more parallelism, in particular if *tokyo* is much slower than *york*. Once again, the client is not penalised: it gets all the guarantees of interest in due time. Definition 8.1.3 captures the new semantics of postconditions.

```
spawn_two_activities (location_1, location_2 : separate LOCATION)
    do
        location_1.do_job
        location_2.do_job
    ensure
        post_1: location_1.is_ready
        post_2: location_2.is_ready
    end

tokyo: separate LOCATION

r (york: separate LOCATION)
    do
        spawn_two_activities (york, tokyo)
        do_local_stuff
        get_result (york)
        do_local_stuff
        get_result (tokyo)
    end
```

Figure 8.3: Separate postconditions

Definition 8.1.3 (Semantics of postconditions) *A postcondition describes the result of a feature's application. Postconditions are evaluated asynchronously; wait by necessity does not apply. Postcondition clauses that do not involve calls on objects handled by the same processors are evaluated independently.*

Following the definition 8.1.2, in the call *spawn_two_activities* (*york*, *tokyo*), the postcondition clause *post_1* is controlled whereas *post_2* is uncontrolled. The client may assume the former (*york*.*is_ready*) after the call because no other client can invalidate it; on the other hand, *tokyo*.*is_ready* cannot be assumed because of the possible interference by other clients. The situation is symmetric to the treatment of preconditions: a controlled precondition clause constitutes an obligation on the client; a controlled postcondition clause is a guarantee given to the client. We use this observation to derive a proof rule for feature calls (see section 8.2 below).

Each query call appearing in a postcondition clause is evaluated by its target's handler; the results of all the queries must be combined into one boolean value. This operation must be performed by the current processor (the one that executed the routine's body) if it is involved; otherwise, any of the involved processors or some "ghost" processor may be used. (Our implementation picks the first processor in the order of appearance; see section 11.3). A violation of a postcondition clause raises an exception in the processor that executed the routine. This gives rise to the problem of asynchronous exceptions mentioned in section 2.2: that processor might

have already left the context of the call, or even become idle in the meantime. This topic is beyond the scope of our work; we only discuss it shortly in section 13.2. See also the exception handling mechanisms proposed by Arslan and Meyer [11], Brooke and Paige [33], and Adrian [2], which tackle the problem.

To wrap up the discussion of postconditions, let's see how their semantics reduces to the traditional sequential semantics if no concurrency is involved. Consider the call *my_x.r (my_a)* in the following code excerpt:

```
-- in class X
r (a: A)
    do
        ...
    ensure
        post_1: a.q
        post_2: q
        post_3: p (a)
    end

-- in class C
my_x: X
my_a: A
 ...
my_x.r (my_a)
do_local_stuff
```

The client, the supplier *my_x*, and the actual argument *my_a* are handled by the same processor. The client cannot proceed to *do_local_stuff* until the postconditions have been evaluated because the client's processor is in charge of evaluating them. As a result, either all the postcondition clauses hold immediately after the call, or an exception is raised. Also, the client can assume all the postconditions of *r* because they are controlled. This corresponds exactly to the sequential semantics.

8.1.3 Invariants

Invariants play an important role in the DbC methodology. They are the primary tool for ensuring the consistence of objects: an instance of class *C* is consistent if and only if it satisfies the invariant of *C*. The invariant must be satisfied in every observable state, i.e. in any state where the object is accessed by a client.

The standard Eiffel semantics applies to class invariants in SCOOP because all the calls appearing in invariants must be non-separate. There is no explicit rule prescribing it; however, the controllability requirement imposed by the call validity rule 6.5.3 can only be satisfied by non-separate entities because invariants have no enclosing routines. The invariant is checked before and after the application of a feature (unless the feature has been called using an unqualified form); an invariant violation raises an exception in the supplier.

```
remove_n ( list : separate LIST [G]; n: INTEGER)
        -- Remove n elements from list .
    require
        list .count >= n
    local
        initial , removed: INTEGER
    do
        from
            initial := list .count
            removed := 0
        until
            removed = n
        invariant
            list .count + removed = initial
        variant
            list .count - initial + n        -- the same as n-removed
        do
            list .remove
            removed := removed + 1
        end
    ensure
        list .count = old list .count - n
    end
```

Figure 8.4: Loop assertions

8.1.4 Loop assertions and check instructions

We apply the asynchronous semantics to loop variants, loop invariants, and **check** instructions. Wait by necessity does not apply; as a result, the evaluation of an assertion is blocking only if the current processor is involved, i.e. the assertion includes non-separate calls. The whole assertion is evaluated at once because there is no possibility of splitting the assertion into individual clauses. All loop variant, invariants, and **check** instructions are *controlled* because all the involved entities serving as targets of calls must be controlled; rule 6.5.3 would be violated otherwise. Therefore, such assertions preserve their contractual character and may be used for formal reasoning. Similarly to postconditions, their asynchronous evaluation does not influence the reasoning style; they may be assumed immediately. If a loop assertion or a check fails, an exception is raised.

Figure 8.4 illustrates the advantages brought by the asynchronous semantics. The assertions appearing in the body of *remove_n* capture the essence of the loop: at every step, the number of elements in *list* is reduced, and the number of remaining elements plus the number of removed elements corresponds to the initial number of elements. Both the variant and the invariant are evaluated asynchronously; the processor handling *list* is asked to evaluate them in due time, i.e. after the previous features targeting *list* but before the next application of *remove*. The request queue of *list* 's handler may look like this:

$$P_{list} : ...[list.remove][eval_{var}][eval_{inv}][list.remove][eval_{var}][eval_{inv}][list.remove]...$$

The client does not wait; it performs the next iteration of the loop, until the exit condition becomes true. Therefore, even if the handler of *list* is very slow, so that the application of *remove* or the evaluation of assertions takes a long time, the client may terminate the execution of *remove_n* knowing that n elements will eventually be removed. (If the call to *remove_n* happens in a context where the actual argument is controlled then the postcondition is controlled and the client may assume it immediately after the call.) Even though the assertions involve separate calls, the correctness and the termination of the loop, as well as the establishment of the postcondition can be proved using the standard sequential reasoning. The new semantics maximises the amount of parallelism while preserving the sequential reasoning style.

Check instructions are treated in the same way as loop assertions; they are evaluated asynchronously in due time. For example, the following sequence of calls:

$x.f$
$x.\ set\_i\ \ (10)$
check $x.i\ =\ 10$ **end**
$x.g$

results in checking $x.i\ =\ 10$ after the application of $x.\ set\_i\ \ (10)$ but before the application of $x.g$. The request queue of x's handler looks like that:

$$P_x : ...[x.f][x.set\_i(10)][eval_{x.i=10}][x.g]...$$

The client is sure that $x.i\ =\ 10$ before the *application* of $x.g$. Note the difference between this guarantee and the guarantee given by an explicit **if** statement involving the query, e.g.

if $x.i\ =\ 10$ **then** $x.g$ **else** ... **end**

Here, due to wait by necessity, the client is sure that $x.i\ =\ 10$ holds before *calling* $x.g$.

8.2 Towards a proof rule

The new semantics of preconditions discussed in section 8.1.1 indicates that a client's responsibilities should be limited to establishing the controlled precondition clauses before the call; the uncontrolled ones may only be established by the environment, i.e. the emergent behaviour of other processors. Similarly, the semantics of postconditions introduced in section 8.1.2 lets clients assume the controlled postcondition clauses after the call; the uncontrolled ones hold when the routine terminates but the client cannot assume them, due to the interference of other clients. We use this observation to define the mutual obligations of clients and suppliers, thus establishing the contractual character of assertions and capturing the essence of DbC in a general (potentially concurrent) context.

Definition 8.2.1 (Mutual obligations of clients and suppliers) *The supplier may assume all the precondition clauses at the the entry to the feature's body; it must establish all postcondition clauses when the body terminates. The client must satisfy all the controlled precondition clauses at the moment of the call; it may assume all the controlled postcondition clauses after the call.*

Although applicable to synchronous and asynchronous calls, the above definition does not use any temporal operators; sequential reasoning applies. This is possible because controlled assertions may be projected in the future, i.e. assumed to hold immediately even if their evaluation

is delayed. From the supplier's point of view, all the preconditions and postconditions are controlled; therefore, its obligations are the same as in a sequential context. The client, however, has fewer obligations; conversely, it gets fewer guarantees. We propose the Hoare-style rule 8.2.2 derived from the sequential proof rule for feature calls but based on the definition 8.2.1. (After a detour through temporal logic [113], it goes back to a style similar to the tentative rule 4.2.1 proposed in OOSC2.) There is no distinction between separate and non-separate assertions; both preserve their contractual character. Only controlled assertion clauses are considered by the client; hence the superscript ctr decorating them in the conclusion of the rule. From the point of view of the supplier, all assertions occurring in Pre_r and $Post_r$ are controlled; that is why all of them appear in the antecedent. The invariant must be fully controlled to avoid side-effects. Direct calls on separate targets are prohibited by the call validity rule but calls of the form $q\ (x)$, where x is separate, must also be prohibited; otherwise, the evaluation of INV might be blocking, and subsequent evaluations might yield different results.

$$\frac{\{INV \wedge Pre_r\}\ \ body_r\ \ \{INV \wedge Post_r\}}{\{Pre_r^{ctr}[\overline{a}/\overline{f}]\}\ \ x.r(\overline{a})\ \ \{Post_r^{ctr}[\overline{a}/\overline{f}]\}} \tag{8.2.2}$$

The sequential-like proof technique 8.2.3 for synchronous and asynchronous feature calls relies on the new proof rule.

Definition 8.2.3 (Proof technique for synchronous and asynchronous calls)
Consider a call $x.r(\overline{a})$. If we can prove that the body of r, started in a state satisfying the precondition, satisfies the postcondition when it terminates, then we can deduce the same property for the above call, with actual arguments $\overline{a}$ substituted for the corresponding formal arguments, every non-qualified call in the assertions (of the form some_property*) replaced by the corresponding property on x (of the form* x.some_property*), and ignoring the uncontrolled assertions.*

Our approach is novel in that it eliminates the need for a special treatment of asynchrony. Sequences of asynchronous calls — or interleaved synchronous and asynchronous calls — may be reasoned about without using temporal operators. (Our earlier solution [113] relied on temporal logic, which resulted in more complex rules.) For example, the correctness of the feature *store_two* in figure 8.5 may be easily demonstrated although its body contains a sequence of separate calls. The precondition of the first call to *buffer .put* is implied by the precondition of *store_two*. Its postcondition is assumed to hold immediately after the call; the postcondition implies the precondition of the second call to *buffer .put*. The postcondition of the second call — again assumed immediately after the call — implies the postcondition of *store_two*. The technique is modular: once we have proved the correctness of *store_two* (which itself requires a proof of {*BUFFER*}.*put*), the same rule may be applied to the calls to *store_two* in clients' code.

Limitations of the proof technique

Rule 8.2.2 is strong enough to prove partial correctness of programs; certain liveness properties, e.g. loop termination, can also be proved. Total correctness, however, cannot be proved because of the potential deadlocks and infinite waiting on non-satisfied uncontrolled preconditions. Recall the York–Tokyo example from figure 8.3. The calls *spawn_two_activities* (*york*, *tokyo*),

```
-- in class C
store_two (buffer : separate BUFFER [INTEGER]; i, j: INTEGER)
        -- Store two elements in buffer .
    require
        buffer .count <= buffer . size - 2
    do
        --   {not buffer . is_full }
        buffer .put (i)
        --   {buffer .count = old buffer .count + 1}
        --   ===>
        --   {buffer .count = old buffer .count + 1 and not buffer . is_full }
        buffer .put (j)
        --   {buffer .count = old buffer .count + 2}
    ensure
        buffer .count = old buffer .count + 2
    end

-- in class BUFFER [G]
put (v: G)
        -- Store v.
    require
        not is_full
    ensure
        count = old count + 1
```

Figure 8.5: Proving asynchronous calls

get_result (*york*), and *get_result* (*tokyo*) happen in a context where *york* is controlled but *tokyo* is not. Therefore, the assertion clauses involving *tokyo* are uncontrolled and they are discarded by the proof rule. Assume that the feature *get_result* is defined as

```
get_result (location : separate LOCATION)
    require
        location . is_ready
    ensure
        location . result_retrieved
```

and the postcondition of *do_local_stuff* is empty. The proof sketch for the routine *r* in figure 8.6 demonstrates that reasoning about *york*'s properties is not hindered; the client has to establish the precondition of *get_result* (*york*) using the postcondition of *spawn_two_actitivies* ; it can also prove the postcondition of *r* using the postcondition of *get_result* (*york*). However, the establishment of the precondition clauses involving *tokyo* is beyond the client's control; it is up to the environment. The client does not need to prove anything but it is not even sure whether the calls involving *tokyo* will ever start executing. The first possibility is that *tokyo*'s handler is never released by its current client; both calls *spawn_two_activities* (*york* , *tokyo*) and *get_result* (*tokyo*) may block for that reason. The second possibility is that *tokyo* becomes available but its state never satisfies the precondition; this may only happen for

```
r (york: separate LOCATION)
    do
        -- {True}
        spawn_two_activities (york, tokyo)
        -- {york.is_ready and True}
        do_local_stuff
        -- {york.is_ready}
        -- ===>
        -- {york.is_ready}
        get_result (york)
        -- {york. result_retrieved }
        do_local_stuff
        -- {york. result_retrieved }
        -- ===>
        -- {york. result_retrieved and True}
        get_result (tokyo)
        -- {york. result_retrieved and True}
    ensure
        york. result_retrieved
    end
```

Figure 8.6: Limitations of the proof technique

get_result (*tokyo*) because it has a precondition. As a result, the total correctness of *r* cannot be proved without considering the emergent behaviour of the whole system; but this requires global reasoning and the use of temporal operators to express such properties as "*tokyo* will eventually become available in a state satisfying *tokyo*. *is_ready*". A non-modular technique for proving the properties of concurrent programs beyond contracts has been proposed by Ostroff et al. [114]. See also the non-modular proof of the producer-consumer example in our earlier paper [113].

8.3 Discussion

8.3.1 Contract redefinition

The proposed semantics of contracts and the proof technique are sound in the presence of polymorphism and dynamic binding; clients are not cheated upon even if contracts are redefined. The standard DbC rules apply: preconditions may be kept or weakened; postconditions and invariants may be kept or strengthened. A redefined precondition results in fewer obligations on the client and (potentially) less waiting at the moment of the feature application; a redefined postcondition gives the client more guarantees. We have already discussed the topic of contract soundness in conjunction with the refined locking mechanism in chapter 7. The support for polymorphism and dynamic binding is discussed in more detail in section 9.1.2.

8.3.2 Importance of lock passing

The proposed proof technique would not be sound without the lock passing mechanism introduced in section 7.2. Consider the proof sketch in figure 8.7. The call $x.\ transfer\_to\ (y)$ is processed synchronously because y is controlled, so a lock passing occurs (see the feature application semantics 6.1.4). Any calls on y within the body of *transfer_to* are guaranteed to be executed before the subsequent call to $y.empty$ issued by the client. Therefore, the postcondition of *transfer_to* may be assumed before the call $y.empty$; it is necessary to prove the correctness of that call (and the whole routine s). If no lock passing occurred, the call $y.empty$ would be processed before the calls issued by the body of *transfer_to* ; the assumption $y.\ is\_full$ would be false and the call $y.empty$ invalid. In fact, following SCOOP_97 rules, x would be able to execute *transfer_to* only after the client terminated s and unlocked y; therefore, $y.\ is\_full$ would eventually become true, which contradicts the promise made by s: its postcondition says that y is empty!

8.3.3 Run-time assertion checking

Meyer [94] mentions the problem of run-time assertion checking in a concurrent context. He concludes

> The assertions are an integral part of the software, whether or not they are enabled at run time. Because in a correct sequential system the assertions will always hold, we may turn off assertion checking for efficiency if we think we have removed all the bugs; but conceptually the assertions are still there. With concurrency the only difference is that certain assertions — the separate precondition clauses — may be violated at run time even for a correct system, and serve as wait conditions. So the assertion monitoring options must not apply to these clauses.

In fact, assertion monitoring may be turned off even for wait conditions. Since the client is responsible for establishing the controlled precondition clauses, the runtime checks of such preconditions are not necessary, provided that the calls have been proved correct (or, at least, we are convinced that they are correct). Uncontrolled precondition clauses must be monitored; however, even here some optimisations are possible. If an uncontrolled precondition clause is satisfied whenever the feature is applied — for example the clause is implied by the class invariant, or maybe it requires some (monotonic) property that has already been established by an earlier call — then there is no point in re-checking the clause every time; we know that it holds.

This issue reveals yet another advantage of the new semantics of postconditions, checks, and loop assertions: the run-time monitoring of assertions has no impact on the semantics of correct programs (except for the incurred overhead). With the traditional sequential semantics, wait by necessity forces the client to wait for the evaluation of an assertion if the run-time checking is turned on but no waiting happens when it is turned off; programs exhibit different behaviour. Our semantics eliminates such inconsistencies.

```
-- in class C
s (x, y: separate X)
    require
        x.is_empty and y.is_empty
    do
        -- {x.is_empty and y.is_empty}
        x. fill
        -- {x. is_full and y.is_empty}
        x. transfer_to (y)
        -- {y. is_full }
        y.empty
        -- {y.is_empty}
    ensure
        y.is_empty
    end

-- in class X
fill
        -- Fill the container.
    require
        is_empty
    ensure
        is_full

empty
        -- Empty the container.
    require
        is_full
    ensure
        is_full

transfer_to (y: separate X)
        -- Transfer the contents to y.
    require
        y.is_empty
    ensure
        y. is_full
```

Figure 8.7: Importance of lock passing

8.4 Related work

The proof technique described here has been largely influenced by the discussions with Jonathan Ostroff and Bertrand Meyer during an informal workshop in May 2006. We have also benefited from the feedback received at the CORDIE'06 conference in York [116].

Ostroff et al. [114] show that contracts provide only a certain measure of correctness but do not guarantee additional safety and liveness properties without global reasoning. The authors model SCOOP_97 programs as fair transition systems, and use temporal logic for describing and proving system properties beyond contractual correctness; their approach is complementary to ours. An assertion testing methodology using the SpecExplorer tool [1] is also presented.

Bailly [16] assumes a different semantics of separate preconditions: they are merely guards of conditional critical regions represented by routine bodies. Guards are excluded from contracts and treated separately from the traditional (correctness) preconditions. The approach does not support inheritance, therefore problems caused by guard strengthening vs. precondition weakening are not discussed. The treatment of postconditions is identical as in the sequential context, although the author comments on the infeasibility of formal reasoning with the sequential proof rule. Other concurrent extensions of Eiffel, such as CEiffel [85, 86], CEE [70], and Distributed Eiffel [62][17] use guard-based synchronisation. Unlike in SCOOP, guards are specified using a different syntax than preconditions.

Sutton [136] describes a new strategy for condition-based process execution, based on delayed evaluation of preconditions and postconditions. Although preconditions have the guard semantics, they are evaluated in parallel with the routine (task) bodies; a task might be allowed to execute even if some of its preconditions have not been evaluated yet. They only have to hold at a particular point of the task's execution; otherwise, the task is put on hold or cancelled. Conversely, the task may terminate even if some of its postconditions have not been established. They have to be established eventually; otherwise, the task must be cancelled (rolled back or compensated). This semantics is similar to ours; postconditions are also "projected" in the future. The precondition semantics proposed by Sutton may be simulated in our model by splitting up a routine into smaller subroutines which require their preconditions to hold on entry to their bodies.

Rodriguez et al. [128] propose a concurrent extension of JML where method guards are treated in a similar way to Sutton's preconditions. Guards are specified using the **when** keyword and they appear in a feature header, after the preconditions. If a feature is called in a state where the guard does not hold, the feature *should* block until the guard is satisfied. Interestingly, the guard does not need to hold at the beginning of the body but only at a point marked with a special statement label **commit**. If no commit point is specified, it is implicitly assumed at the end of the body. No implicit waiting mechanism is provided; it is up to the programmer to implement it, e.g. in the form of a busy-waiting loop. Therefore, we can view guards as a help in the static verification of atomicity properties but they do not facilitate the construction of concurrent programs — programmers are still forced to write synchronisation code by hand. This inevitably leads to the occurrence of inheritance anomalies. The generalised semantics of preconditions that we propose avoids this problem (see section 10.5).

Traditionally, proof methods for concurrent programs are non-compositional, i.e. it is necessary to consider the whole program in order to prove correctness of its parts. In his PhD

[1] http://research.microsoft.com/SpecExplorer

dissertation [71], Cliff Jones describes a compositional approach to proving correctness properties of concurrent shared-memory programs. He enriches contracts with two additional assertions — *rely* and *guarantee* — that represent assumptions on the environment of a process and commitment to the environment, respectively. Such assertions are used together with the standard preconditions and postconditions. Rely–guarantee specifications may only be applied to shared-memory models with no aliasing but similar approaches (*assumption–commitment*) have been proposed for message-passing systems [98]. These are more appropriate for SCOOP-like models that are based on asynchronous feature calls. An interesting survey of research efforts related to compositional approaches for concurrency is [72]. A fully modular proof system for SCOOP would require much more expressive contracts: new types of assertions would be necessary to capture the locking behavior of routines (i.e. what additional resources a routine may request during the execution of its body) and their frame properties. These may be viewed as a particular case of assumption–commitment specifications.

9

Advanced object-oriented mechanisms in SCOOP

THE interplay between advanced object-oriented mechanisms and concurrency is of paramount importance for SCOOP. The basic model presented so far excluded genericity, agents, and once functions; we omitted these features to keep the formal model as compact as possible and facilitate its understanding. Also missing was a detailed discussion of inheritance, polymorphism, and their relation with other mechanisms. These issues are vital in practice; an appropriate support is needed for all the above mechanisms.

For each mechanisms, we discuss its interaction with concurrency, point out the specific problems raised in the context of SCOOP, and propose an appropriate way to integrate the mechanism with our framework. Where necessary, we provide a set of validity rules which refine the type system introduced in chapter 6.

9.1 Inheritance and polymorphism

Inheritance (including its *multiple* and *repeated* variants) enables a stepwise specialisation and refinement of abstractions implemented by classes. Polymorphism lets entities of type T represent objects of any type U that conforms to T. These two mechanisms constitute the backbone of the object-technology; they foster the modularity, extendibility, and reusability of code, by providing a convenient support for abstraction (see chapter 1). Abstraction enables stripping away irrelevant details, e.g. how a given feature is implemented. Usually, an abstract concept is captured in a base class whose descendants provide different specialisations and implementations; these concrete implementations may be "plugged in" where necessary, using polymorphic attachments and dynamic binding.

9.1.1 Multiple inheritance

SCOOP does not alter the inheritance mechanism; all kinds of inheritance — single, multiple, repeated — are permitted; the standard Eiffel rules apply. The type system relies on class types and their hierarchy but the concurrency-relevant parts — processor tags — do not interfere with class types. (We do not distinguish "separate" and "non-separate" classes.) Therefore, no additional restrictions are placed on class hierarchies; the usual rules for feature introduction, renaming, join, and undefinition apply (see the Eiffel standard [53]). There is, however, one new syntactic element which has to be taken care of by the inheritance rules: the unqualified proces-

sor tag. Such processor tags are used in the enriched type system to mark entities representing objects handled by the same processor, e.g.

```
x: separate <px> X
y: separate <px> X
```

The tag *px* must be declared in the same class; an attribute-like declaration

```
px: PROCESSOR
```

is used. Processor tags must be unique: no other tag, entity, or feature within the same class may be called *px*. This raises the problem of potential name clashes in the context of inheritance. We need to extend the rules for renaming and join so that they also apply to processor tags. A processor tag may now be renamed in a descendant, e.g.

```
class A
feature
    px: PROCESSOR
    x, y: separate <px> X
    ...
end

class B
inherit A rename px as pt end    -- Renaming prevents a name clash.
feature
    z: separate <pt> Z
    px: INTEGER    -- A new feature is called px.
    ...
end
```

Naturally, the renaming of *px* influences all the type annotations involving that tag, i.e. the inherited features *x* and *y* in class *B* have the processor tag *pt*.

When inheriting from several classes that declare the same processor tag, it is possible to keep the tags distinct (through renaming) or arrange for a join, e.g.

```
class A
feature
    px: PROCESSOR
    py: PROCESSOR
    ...
end

class B
feature
    px: PROCESSOR
    py: PROCESSOR
    ...
end

class C
inherit
    A
    B rename py as pt end
...
end
```

Class *C* joins both tags *px* inherited from *A* and *B*; processor tags are treated similarly to deferred features, i.e. the join operation does not require renaming or undefining any of them. Tags *py*

are kept distinct by renaming one of them as *pt*.

Since processor tags serve as type annotations but they are not features, their redefinition or undefinition is prohibited.

9.1.2 Polymorphism and dynamic binding

The SCOOP framework fully supports polymorphism: the refined validity rules ensure the type safety of operations on polymorphic entities and dynamically bound features. Clients are not deceived, i.e. all the redefined versions of a feature abide by the original contract known to the client. This is achieved through a combination of several elements: the conformance rules for enriched types, the validity rules for feature redefinition, and the new semantics of contracts.

The subtype relation defined in figure 6.20 stipulates that a type $U = (\delta, \beta, Y)$ is a subtype of $T = (\gamma, \alpha, X)$ if and only if the three components of U — the detachable tag δ, the processor tag β, and the class type Y — conform to the corresponding components of T. The conformance relation on detachable tags says that attached (!) conforms to detachable (?), e.g. $(!, \bullet, X)$ is a subtype of $(?, \bullet, X)$, and $(!, \top, X)$ is a subtype of $(?, \top, X)$. The conformance relation on class types is defined by the rules of sequential Eiffel [53]. Essentially, it follows the inheritance relation: *D* conforms to *C* if *D* inherits from *C*. Special rules apply to expanded classes and generic derivations. *D* conforms to an expanded class *EC* if and only if *D* = *EC*. The conformance of generic class derivations is discussed in section 9.2 below. Processor tags form a lattice with the top element '$\top$', the bottom element '$\bot$', and the other elements — '$\bullet$', unqualified tags, and qualified tags — placed between the two, so that they conform to '$\top$' but not to each other.

The signature conformance rule of Eiffel (rule 8.14.4 /VNCS/ in [53]) permits covariant redefinition of argument types, provided that the redefined arguments are declared as detachable. The rule for result types is less strict: the redefined type simply has to conform to the original one. In chapter 7 we have introduced a stronger rule (7.1.3) which restricts the redefinition on formal argument types so that only arguments not appearing as targets of calls in the inherited postconditions may be redefined; the redefinition is covariant for class type (i.e. from a superclass to a subclass) like in the sequential Eiffel) but goes in the opposite direction in the detachable tag, i.e. an attached argument may be redefined into a detachable one but not vice-versa. The rule puts no additional restrictions of the redefinition of result types.

Redefinition of processor tags

Class types and detachable tags are taken care of by the rules /VNCS/ and 7.1.3; we still need to clarify the rules for processor tags. Figure 9.1 illustrates the effects of result type redefinition. The attribute *x* is redefined in class *B* into a non-separate one; its processor tag changes from '$\top$' to '$\bullet$'. The attribute *y* is redefined from non-separate into separate, i.e. its processor tag changes from '$\bullet$' to '$\top$'. After the polymorphic assignment *a* := *b*, features *r* and *s* are called with the dynamically bound actual arguments of the redefined type. Feature *r* expects an argument of type $(!, \top, X)$ but receives an argument of type $(!, \bullet, X)$. The formal argument *x* is properly locked following the locking semantics of attached types introduced in chapter 7; here, locking is trivial because the actual argument is handled by the current processor. Because the actual argument is non-separate, the precondition *x*.*some_requirement* is controlled, i.e. a

contract violation occurs if it does not hold immediately (see section 8.1.1). The call $x.f$ in the body of r is also valid because x is controlled in that context (see rule 6.5.3). The postcondition $x.some\_guarantee$ is evaluated synchronously, like in a sequential context. Therefore, the redefinition of attributes from less specific to more specific types is sound. (Recall that it was not the case in SCOOP_97; see section 5.10.) On the other hand, the redefinition of attributes from more specific to more general types is not sound because it introduces potential traitors. The call $s\ (a.y)$ passes a separate actual argument to s which expects a non-separate one. No appropriate locking is performed because y is assumed to be trivially locked; hence y becomes a traitor: $y.f$ is executed without respecting the atomicity requirements. Furthermore, the semantics of contracts is applied incorrectly: the precondition $y.some\_requirement$ and the postcondition $y.some\_guarantee$ are assumed to be controlled, which may lead to spurious contract violations. Therefore, such a redefinition of result types should be prohibited.

Figure 9.2 illustrates the effects of a formal argument redefinition. The original version of r in class A takes a separate formal argument; the original version of s takes a non-separate one. Both features are redefined in class B to take a non-separate and a separate argument respectively. After the polymorphic assignment $a := b$, the dynamically bound versions of r and s are called with the actual arguments of the type required by the original features. As a result, the executed version of r expects an argument of type $(!, \bullet, X)$ but receives an argument of type $(!, \top, X)$. The formal argument x becomes a traitor: it is assumed to be handled by the current processor, so no actual locking occurs; the call $x.f$ is executed without respecting the atomicity requirements. The assignment $my\_y := x$ is permitted, which results in the introduction of yet another traitor. Additionally, the inherited precondition $x.some\_requirement$ is incorrectly viewed as controlled, thus reduces to a one-off check; similarly, the postcondition is viewed as controlled and evaluated synchronously, which may cause a deadlock. Therefore, a redefinition of the processor tag of argument types from less specific to more specific must be prohibited. On the other hand, a redefinition in the opposite direction does not cause any trouble: the new version of s gets an actual argument of type $(!, \bullet, X)$ conforming to the expected type $(!, \top, X)$. Consequently, its formal argument x is properly locked (because the actual argument is handled by the current processor); the precondition and the postcondition correctly reduce to one-off checks.

The examples above only use two processor tags: '$\top$' and '$\bullet$'. Nevertheless, the results of our analysis apply to other processor tags as well; any unqualified processor tag may be taken instead of '$\bullet$'. If the original attribute x: **separate** X in figure 9.1 was redefined into a more specific x: **separate** $<px>$ X, the polymorphic call $r\ (a.x)$ would not cause any problems. On the other hand, a redefinition of y: **separate** $<py>$ Y into a more general y: **separate** Y would lead to similar problems as before: the call $s\ (a.y)$ would result in an atomicity violation, and the semantics of contracts would be applied incorrectly, leading to the incorrect synchronisation and a potential deadlock. Similar effects may be observed when using an explicit processor tag in the formal argument of any routine in figure 9.2. A redefinition of r's argument from x: **separate** X to x: **separate** $<px>$ X could introduce traitors, e.g. an assignment from x to another entity of type **separate** $<px>$ X would be possible although the object represented by x is not necessarily handled by px. A redefinition of s's argument from x: **separate** $<px>$ X to x: **separate** X causes no trouble at all. Rule 9.1.1 expresses succinctly the correctness criteria for the redefinition of processor tags.

Definition 9.1.1 (Feature redefinition rule (processor tags)) *A result type may be redefined from separate (with processor tag '$\top$') to more specific. A formal argument may be redefined*

```
class A
feature
    x: separate X
    y: Y
     ...
end

class B inherit A redefine x, y end
feature
    x: X
    y: separate Y
     ...
end

-- in class C
r (x: separate X)
    require
        x.some_requirement
    do
        x.f
    ensure
        x.some_guarantee
    end

s (y: Y)
    require
        y.some_requirement
    do
        y.f                    -- y is a traitor
    ensure
        y.some_guarantee
    end

a: A
b: B
 ...
a := b        -- Polymorphic assignment
r (a.x)       -- Valid
s (a.y)       -- Problematic
```

Figure 9.1: Redefinition of result types

to separate (with processor tag '⊤').

Unified feature redefinition rule

The redefinition rules for the individual components of a type are now clarified:

- *Class types* may be redefined covariantly both in result types and argument types but the redefinition of a formal argument forces it to become detachable (rule /VNCS/). For example, assuming that *Y* conforms to *X*, an argument *x*: *X* may be redefined into *x*: ?*Y* but not *x*: *Y*. An attribute may be redefined from *my_x*: *X* into *my_x*: *Y*.
- *Detachable tags* may be redefined from '?' to '!' in result types. They may be changed

```
class A
feature
    r (x: separate X)
        require
            x.some_requirement
        do
            x.f
        ensure
            x.some_guarantee
        end

    s (x: X)
        require
            x.some_requirement
        do
            x.f
        ensure
            x.some_guarantee
        end
end

-- in class C
a: A
b: B
my_sep_x: separate X
my_x, my_y: X
 ...
a := b              -- Polymorphic assignment
a.r (my_sep_x)      -- Problematic
a.s (my_x)          -- Valid
```

```
class B
inherit A redefine r, s end
feature
    r (x: X)
        do
            x.f             -- traitor
            my_y := x       -- traitor
        end

    s (x: separate X)
        do
            x.f
        end
end
```

Figure 9.2: Redefinition of argument types

from '!' to '?' in argument types, provided that no call on the redefined argument occurs in the original postcondition (rule 7.1.3).

- *Processor tags* may be redefined from '⊤' to something more specific in result types, and from more specific to '⊤' in argument types (rule 9.1.1).

A unified rule should ensure the soundness of feature redefinitions. We understand soundness as preservation of type safety and compatibility of the synchronisation requirements of the redefined feature with those expressed by the original contract. More precisely, a redefinition must not introduce potential traitors; it must ensure the necessary locking and condition synchronisation without causing any additional waiting on the clients' side. The rule for result types is straightforward because all three type components may be redefined from less specific to more specific. Therefore, we may use directly the subtype relation defined in section 6.11. The situation is more complex for argument types: two type components may be redefined from

more specific to more general, whereas the third one may change in the opposite direction. As a result, no subtype relation can be established between the original and the redefined type: if a type T is replaced by U, neither $U \preceq T$ nor $T \preceq U$ need to hold.

Definition 9.1.2 (Feature redefinition rule) *The result type of a feature may be redefined from T to U if and only if $U \preceq T$. The type of a formal argument x may be redefined from $T = (\gamma, \alpha, X)$ to $U = (\delta, \beta, Y)$ if and only if the following conditions hold:*

1. *Y conforms to X.*
2. *If T is detachable or $Y \neq X$ then U is detachable.*
3. *If U is detachable then no calls on x occur in the original postcondition.*
4. *U and T have identical processor tags ($\beta = \alpha$); otherwise, U is separate ($\beta = \top$).*

Compatibility with DbC

Thanks to the unified contract semantics and the *inherited precondition rule* 7.1.4, which says that all inherited precondition clauses involving calls on a detachable formal argument are trivially true, the feature redefinition rule 9.1.2 is compatible with DbC: preconditions may only be kept or weakened; postconditions may only be kept or strengthened. Figure 9.3 illustrates it: routine *r* is redefined so that the type of its formal argument *x* changes from $(!, \bullet, X)$ to $(!, \top, X)$, its argument *y* is redefined from $(!, \bullet,)$ to $(?, \top, Z)$, and its result type from $(?, \bullet, Z)$ to $(!, \bullet, Z)$; we assume that *Z* inherits from *Y*. The new signature of *r* conforms to the original one as required by the rule 9.1.2. The redefined precondition

$(x.\mathit{is\_empty}$ **and** $y.\mathit{is\_empty})$ **or else** $x.\mathit{count} = 1$

reduces to

$(x.\mathit{is\_empty}$ **and True**$)$ **or else** $x.\mathit{count} = 1$

following the rule 7.1.4 (*y* is detachable), hence to

$x.\mathit{is\_empty}$ **or else** $x.\mathit{count} = 1$

It is weaker than the original precondition because

$(x.\mathit{is\_empty}$ **and** $y.\mathit{is\_empty})$ ===> $(x.\mathit{is\_empty}$ **or else** $x.\mathit{count} = 1)$

So, the client's obligations are not strengthened. The redefined postcondition

$x.p\ (y)$ **and then** $(x.\mathit{count} > 5$ **and Result**$.\mathit{is\_empty})$

clearly implies the original postcondition $x.p\ (y)$. Therefore, the guarantees given to the client are at least as strong as promised by the original contract. Note that *x* cannot be redefined into detachable (clause 3 of 9.1.2) because it occurs as target of a call in the original postcondition; *y* can become detachable because there are no calls on *y* in the original postcondition. In fact, *y* must become detachable because its class type has changed covariantly (clause 2 of 9.1.2). The result type has become attached; consequently, calls on **Result** may appear in the postcondition (and the body) of *r*; such calls would be invalid in the original version of *r*.

```
class A
feature
    f (x: X; y: Y): ?Z
        require
            x.is_empty
            y.is_empty
        ensure
            x.p (y)
end

class B inherit A redefine r end
feature
    r (x: separate X; y: ?separate Z): Z
        require else
            x.count = 1
        ensure then
            x.count > 5
            Result.is_empty
end
```

Figure 9.3: Redefinition of contracts

Precursor calls

The redefinition of argument types may lead to invalid **Precursor** calls. This problem has already been identified (for detachable tags) in section 7.1.3; unfortunately, it may also be caused by the redefinition of an argument into a separate one, and by a covariant redefinition of an argument's class type (because it forces the argument to become detachable). Figure 9.4 illustrates both cases: the redefined version of *s* takes a separate argument *x* and a detachable argument *y*, and performs a precursor call **Precursor** (x, y) which is invalid because the actual argument types — $(!, \top, X)$ and $(?, \bullet, Z)$ — do not conform to the expected ones: $(!, \bullet, X)$ and $(!, \top, Y)$ respectively. The *object test* mechanism (see section 6.7) offers a simple albeit verbose solution: before performing a call, the actual arguments are downcast to the types required by the original routine. An object test is required for each problematic argument; hence a cascade of object tests wrapping the actual precursor call.

9.1.3 Deferred classes

An Eiffel class may be declared as **deferred**, i.e. not fully implemented. A deferred class may have zero or more deferred features; such features are specified but not implemented. For example, the class *COMPARABLE* in figure 9.5 has one deferred feature $<$; other features are effective, i.e. fully implemented (only one of them, $<=$, is shown here). Deferred classes cannot be instantiated but their effective descendants can.

Deferred classes are useful for analysis and design because they make it possible to capture the essential aspects of a system while leaving details to a later stage. They are an important tool for structuring class hierarchies. Also, generic constraints are often expressed in terms of

```
class A
feature
    s (x: X; y: separate Y)
        do
            x.f
        end
end

class B inherit A redefine s end
feature
    s (x: separate X; ?separate Z)   -- Assume Z conforms to Y
        do
            Precursor (x, y)          -- Invalid
            if {aux_x: X} x and then {aux_y: separate Y} y then
                    Precursor (aux_x, aux_y)      -- Valid
            end
            ...
        end
end
```

Figure 9.4: Problematic **Precursor** calls

deferred class types. Since the type system clearly separates the notions of locality (expressed through processor tags) and class types, SCOOP_97's restrictions on the use of deferred classes (see section 5.12.2) go away. Deferred classes may now be freely used to construct separate types, as illustrated in figure 9.5, thus enriching the support for polymorphism and genericity.

9.2 Genericity

So far, the rules of our type system only considered simple class types; we have used generic types in several examples but have not provided a formal justification for their safety in a concurrent context. This section discusses the use of genericity in SCOOP and refines the conformance and validity rules to accommodate the mechanism.

Genericity gives us a new level of flexibility through *type parameterisation*. A class may be defined as generic, e.g. *LIST* [*G*], yielding more than one class type: *LIST* [*INTEGER*], *LIST* [*FIGURE*] and so on, parameterised by G. (These class types are referred to as *generically derived* types, or *generic types*.) Two forms of genericity are available:

- *Unconstrained genericity*: G represents an arbitrary type.
- *Constrained genericity*: one can demand certain properties of the types represented by G, enabling a more specific use of G in the class text.

SCOOP supports both forms of genericity. Separate annotations may appear in actual and formal generic parameters, as well as in constraints; their use is only limited by the conformance

```
deferred class COMPARABLE
 ...
feature
    infix ''<'' (other: like Current): BOOLEAN
            -- Is current object less than other?
        deferred
        ensure then
            asymmetric: Result implies not (other < Current)
        end

    infix ''<='' (other: like Current): BOOLEAN
            -- Is current object less than or equal to other?
        do
            Result := not (other < Current)
        ensure then
            definition : Result = (Current < other) or is_equal (other)
        end
    ...
end

class E [G -> separate COMPARABLE]    -- generic constraint
    ...
end

-- in class C
my_d: separate COMPARABLE              -- separate and deferred
f (d: separate COMPARABLE) do ... end  -- separate and deferred
```

Figure 9.5: Use of deferred classes

rules of the type system. Additionally, it is possible to construct arbitrarily nested types with separate actual generic parameters, e.g.

```
l: separate LIST [HASH_TABLE [separate SET [STACK [separate A [...]]]], STRING]
```

9.2.1 Generic parameters

All actual generic parameters appearing in a type must conform to the corresponding formal parameters declared in the type's base class (see the generic derivation rule 8.12.11 /VTGD/ in [53]). The conformance relation on SCOOP types stipulates that a type U is a subtype of T only if the processor tag of U conforms to the processor tag of T. For example, $(!, \bullet, X)$ and $(!, px, X)$ conform to $(!, \top, X)$ but not the other way round. Consequently, separate actual generic parameters may only be used if the corresponding formals are themselves separate.

An unconstrained formal generic parameter allows actual parameters of an arbitrary type. In sequential Eiffel, an arbitrary type conforms to *ANY* because *ANY* is the most general type. (In fact, the most general type is ?*ANY* but we omit the detachable tag for the moment; its

use in generic types is discussed in section 9.2.5 below.) In SCOOP, the most general type (ignoring detachability again) is separate: $(!, \top, ANY)$. Therefore, an unconstrained formal generic parameter is implicitly constrained to that type, i.e. a declaration

class *LIST* [*G*] ... **end**

is equivalent to

class *LIST* [*G* −> **separate** *ANY*] ... **end**

The Eiffel rule for constraining types (8.12.7 in [53]) needs to be refined as follows:

Definition 9.2.1 (Constraint, constraining types of a formal generic parameter) *The constraint of a formal generic parameter is its* constraint_part *if present, and otherwise* ***separate*** *ANY. Its constraining types are all the types listed in its* constraining_types *if present, and otherwise just* $(!, \top, ANY)$.

This refinement has no impact on the clients using a generic type, e.g. the type *LIST* [*X*] which is valid in sequential Eiffel for an arbitrary class *X*, remains valid in SCOOP as well. On the other hand, the separateness of a formal generic parameter puts some restrictions on its use in the class text. For example, the class *A* [*G*] in figure 9.6 is valid in Eiffel but not in SCOOP because the call *item*. *is_equal* (*v*) appears in a context where *item* is not necessarily controlled. Since *item* is (potentially) separate but *f* does not lock its handler, the call is invalid. In practice, however, the restriction imposed by the new rule is not problematic; for example, all the generic classes in EiffelBase remain valid. Also, not all the calls on entities of type *G* are invalid, e.g. a potentially separate call *v*. *is_equal* (*item*) is correct because its target is controlled: *v* appears as formal argument of *f* so its handler is locked by *f* if necessary.

```
class A [G]
feature
    item: G

    f (v: G)
        do
            if item.is_equal (v) then    -- Invalid
                ...
            end
            if v.is_equal (item) then    -- Valid
                ...
            end
        end
end
```

Figure 9.6: Incorrect use of unconstrained generic parameter

9.2.2 Constrained genericity

A formal generic parameter may be explicitly constrained, e.g.

```
class A [G -> separate X] ... end
```

or

```
class B [G -> X] ... end
```

Since any actual parameter must conform to the formal, only the types conforming to the constraining type may be used as actual generic parameters, e.g.

```
a1: A [separate Y]      -- Valid if Y conforms to X
a2: A [X]
a3: A [separate ANY]    -- Invalid
b1: B [Y]
b2: B [separate X]      -- Invalid
```

A [**separate** *X*] is invalid because the actual parameter type $(!, \top, ANY)$ does not conform to $(!, \top, X)$; similarly, *B* [**separate** *X*] is invalid because $(!, \top, X)$ does not conform to $(!, \bullet, X)$. The latter example suggests a way to enforce the non-separateness of actual parameters: it suffices to constrain the corresponding formal parameter to $(!, \bullet, ANY)$, e.g.

```
class C [G -> ANY] ... end
```

A constraint gives additional information on possible actual types represented by *G*; for example, if the base class of the constraining type has feature *f* then every actual parameter type is guaranteed to have it as well. This lets the implementer of the generic class apply *f* to entities of type *G* appearing in the class text. A typical example is the constraint based on the class *COMPARABLE*, depicted in figure 9.7. *COMPARABLE* provides a number of comparison op-

```
class D [G -> separate COMPARABLE]
feature
    item: G

    r (v: G)
        require
            v <= item              -- Valid
        do
            if v < item then       -- Valid
                ...
            else
                ...
            end
        end
end
```

Figure 9.7: Constrained genericity

erators, e.g. '<', '<=', '>'; they may be freely used on entities of type *G* within the class text of *D*. Both the precondition of *r* and the expression *v* < *item* appearing in its body are valid.

The above example also illustrates nicely the interplay of genericity with the new semantics of attached types and contracts: independently of whether the actual generic parameter *G* is

separate or not, the feature *r* ensures an appropriate synchronisation; if necessary, the execution of *r* is delayed until *v*'s handler is locked and the precondition holds.

9.2.3 Actual result types

The locality of an actual generic parameter is always relative to the instance of the class where the corresponding formal generic parameter is declared, e.g. *l*: *LIST* [**separate** *BOOK*] denotes a non-separate list of separate books, i.e. the elements of *l* are separate from it (see figure 9.8). On the other hand, *sl*: **separate** *LIST* [*BOOK*] is separate from the current object; the elements of *sl* are handled by the same processor as *sl* (see figure 9.9).

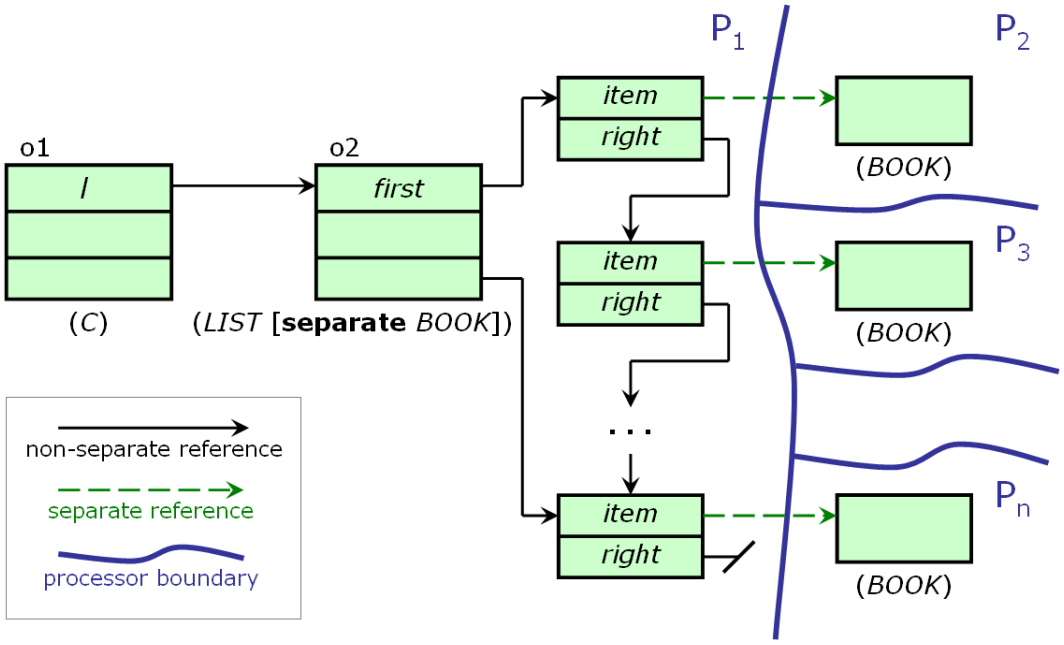


Figure 9.8: *LIST* [**separate** *BOOK*]

If the locality of the elements is relative to the container and not to the client using the container, then what is the type of the elements as seen by the client? (We talk about containers here but the discussion concerns all generic classes, not only those representing containers.) The type combinator '$\star$' is used to compute it: the type of the container (as seen by the client) is combined with the type of the elements (as seen by the container). Applying this technique to the examples above yields the following results: the elements of *l* have the type $(!, \top, BOOK)$ because

$$(!, \bullet, LIST[...]) \star (!, \top, BOOK) = (!, \top, BOOK)$$

and the elements of *sl* have the type $(!, \top, BOOK)$ because

$$(!, \top, LIST[...]) \star (!, \bullet, BOOK) = (!, \top, BOOK)$$

The calculated types correspond to those returned by the calls to features of a generic parameter type, e.g. *item* and *first* . (These features are declared as *item*: *G* and *first* : *G* in the class *LIST* [*G*]; substituting *G* with with the actual generic parameter and combining the type of the container with the obtained result type yields $(!, \top, BOOK)$.) Therefore, an entity used for storing an element retrieved from *l* or *sl* needs to be declared accordingly:

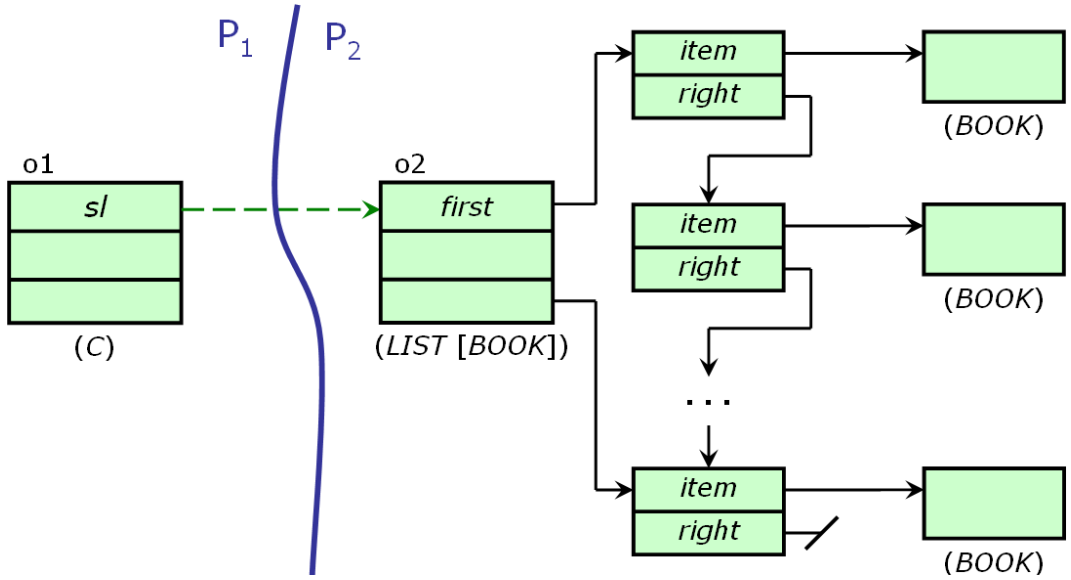


Figure 9.9: **separate** *LIST* [*BOOK*]

my_book: **separate** *BOOK*
...
my_book := *l* . *first*
my_book := *sl* . *item*

The combinator '$\star$' may be applied recursively (just like in multi-dot expressions) to nested generic types, e.g.

ml: *LIST* [**separate** *ARRAY* [*STACK* [**separate** *BOOK*]]]
...
my_book := *ml*.*item* . *item*(4) . *top*

The innermost type in the declaration of *ml* is seen by the client as $(!, \top, BOOK)$ because

$$(!, \bullet, LIST[...]) \star (!, \top, ARRAY[...]) \star (!, \bullet, STACK[...]) \star (!, \top, BOOK) =$$
$$(!, \top, ARRAY[...]) \star (!, \bullet, STACK[...]) \star (!, \top, BOOK) =$$
$$(!, \top, STACK[...]) \star (!, \top, BOOK) =$$
$$(!, \top, BOOK)$$

Consequently, the expression *ml*.*item* . *item*(4) . *top* has the type $(!, \top, BOOK)$ and the above assignment to *my_book* is valid if and only if its target is declared as

my_book: **separate** *BOOK*

or with a supertype of $(!, \top, BOOK)$, e.g.

my_book: **separate** *ANY*

9.2.4 Actual argument types

The type of an actual argument must conform to the type of the corresponding formal in the called feature, after an appropriate combination (using the operator '$\otimes$') with the type of the

target on which the feature is called. If the type of the formal argument is a generic parameter, then the corresponding actual parameter is used in its place. For example, if the feature *put* in the class *LIST* [*G*] expects an argument of type *G*, then the actual argument *my_book* of the call

l: *LIST* [**separate** *BOOK*]
...
l.*put* (*my_book*)

has to conform to the type $(!, \top, BOOK)$ because

$$(!, \bullet, LIST[...]) \otimes (!, \top, BOOK) = (!, \top, BOOK)$$

It is therefore sufficient to declare it as

my_book: **separate** *BOOK*

If the generic type is nested, e.g.

nl: **separate** <*px*> *LIST* [*ARRAY* [*STACK* [*BOOK*]]]
...
nl.*item*.*item*(3).*put* (*my_book*)

we follow the same scheme; here, the actual argument must conform to $(!, px, BOOK)$ because the target *nl*.*item*.*item*(3) has the type $(!, px, STACK[BOOK])$ — obtained using the combinator '$\star$' — and *put* takes an argument of type G (a formal generic parameter, hence replaced by the actual parameter $(!, \bullet, BOOK)$), and

$$\begin{aligned}((!, px, LIST[...]) \star (!, \bullet, ARRAY[...]) \star (!, \bullet, STACK[BOOK])) \otimes (!, \bullet, BOOK) = \\ (!, px, STACK[BOOK]) \otimes (!, \bullet, BOOK) = \\ (!, px, BOOK)\end{aligned}$$

Now, *my_book* has to be declared as

my_book: **separate** <*px*> *BOOK*

or with a subtype of $(!, px, BOOK)$.

9.2.5 Detachable generic parameters

So far, we have focussed on processor tags and class types, and ignored detachable tags. Beyond their usual meaning — marking entities as potentially void — detachable tags play a particular role in the genericity mechanism: they impose certain restrictions on type conformance and ensure the soundness of covariant subtyping, thus eliminating catcalls (see section 9.2.6 below). The detachable tag '?' decorating a formal generic parameter, e.g.

class *A* [?*G*] ... **end**

turns it into a *self-initialising* parameter; valid actual generic parameters must be either detachable — in this particular case conform to $(?, \top, ANY)$ because ?*G* is equivalent to ?*G* −> **separate** *ANY* — or be themselves *self-initialising*, i.e. list *default_create* from class *ANY* among their creation procedures (rule 8.12.11 /VTGD/). For example, the following declarations are valid:

a1: *A* [?**separate** *BOOK*]
a2: *A* [?*ANY*]
a3: *A* [*INTEGER*]

because the actual parameters satisfy the above requirement: $(?, \top, BOOK) \preceq (?, \top, ANY)$, $(?, \bullet, ANY) \preceq (?, \top, ANY)$, $(!, \bullet, INTEGER) \preceq (?, \top, ANY)$ and *INTEGER* is self-initialising. On the other hand, the declaration

a4: *A* [**separate** *BOOK*]

is invalid because the actual parameter $(!, \top, ANY)$ is not self-initialising: *BOOK* does not list *default_create* as creation procedure. Self-initialising formal generic parameters may be explicitly constrained, e.g.

class *B* [?*G* −> **separate** *X*] ... **end**
class *D* [?*G* −> *X*] ... **end**

Actual generic parameters must conform to the constraining type and, if attached, be self-initialising as required by the above rule for detachable tags.

A note on the syntax: rather than splitting the type annotation so that one part ('?') appears before the parameter while the rest comes after it, as in ?*G* −> **separate** *X*, one could write the whole type in the constraint: *G* −> ?**separate** *X*. Nevertheless, we opt for the former syntax because we want to preserve the compatibility with the sequential Eiffel which uses the form ?*G*. Furthermore, this syntax is also more compact: to declare a parameter as self-initialising, it suffices to write ?*G*; with the alternative syntax, an explicit constraint is needed: *G* −> ? **separate** *ANY*.

9.2.6 Type conformance

Now that the basic rules of conformance between actual and formal generic parameters have been clarified, let's have a look at their combination with inheritance and polymorphism. There are two issues of interest:

- *Inheritance of generic classes*
  Should the specialisation of a separate parameter into a non-separate one be allowed, i.e.

```
class A [G]        -- G -> separate ANY
...
end

class B [G -> ANY]
inherit A [G]
...
end
```

  or the other way round

```
class A [G -> ANY]
...
end
```

```
class B [G]      -- G -> separate ANY
inherit A [G]
 ...
end
```

- *Conformance of generic types*
  Does *A* [*X*] conform to *A* [**separate** *X*] or the other way round?

Inheritance of generic classes

Figure 9.10 shows a problematic scenario involving inheritance and constrained genericity. Class *B* inherits from *A* and strengthens (makes more specific) the constraint on the generic parameter from $(!, \top, ANY)$ to $(!, \bullet, ANY)$. Since the type *G* is now more specific, one may do the same things with entities of this type in *B* as in *A*, and possibly more. As a result, all the features inherited from *A* remain valid in *B*. Features defined in *B* may do more sophisticated things, e.g. *r* performs a call to *item*.*f* that would be incorrect in *A*; in *B*, however, *item* is guaranteed to be non-separate (and attached), it is therefore controlled and the call is valid. On the other hand, the same call becomes invalid in class *D* because *D* relaxes the constraint on *G*; now, *item* may be separate and its base class does not necessarily conform to *Y* (or even to *X*). Therefore, *item*.*f* is doubly invalid: because its target is not controlled, and because the call is a catcall as *f* is not necessarily defined for *item*. The source of the problem is the non-conformance of the new constraint to the inherited one: $(!, \top, ANY) \not\leq (!, \bullet, Y)$.

```
class A [G -> separate X]
feature
    item: G
     ...
end

class B [G -> Y]    -- Y inherits from X
inherit A [G]
    r   do
            item.f    -- Valid
        end
end

class D [G]         -- G -> separate ANY
inherit B [G]
    -- r is invalid here
end
```

Figure 9.10: Constrained genericity and inheritance

Since a non-conforming constraint violates the type safety, the inheritance relation between generic classes must preserve the constraints, i.e. keep or strengthen them, as expressed by the rule 9.2.2 below.

Definition 9.2.2 (Inheritance of classes with constrained generic parameters) *If a formal generic parameter G is constrained to a type T in the ancestor class C, then in any class that inherits from C, G must be constrained to a type U such that $U \preceq T$. Additionally, if G is self-initialising in C then it must be self-initialising in all descendants of C.*

We rely on the standard definition of type conformance; however, the second part of the rule reflects the special semantics of detachable tags discussed in section 9.2.5. It is prohibited to redefine a formal generic parameter from self-initialising to non-self-initialising; figure 9.11 illustrates the reason for this restriction. Class *B* inherits from *A* and constrains the formal generic parameter to $(?, \bullet, X)$, keeping it self-initialising like in the ancestor class. The inherited routine *g* is valid because its result (**Void**) conforms to the constraint: $(?, \bot, NONE) \preceq (?, \bullet, X)$. In class *D*, which also inherits from *A* but constrains *G* to $(!, \bullet, X)$, thus making it non-self-initialising, the routine *g* is invalid because its result does not conform to the constraint: $(?, \bot, NONE) \not\preceq (!, \bullet, X)$. Hence the requirement to preserve the self-initialising character of inherited formal generic parameters.

```
class A [?G -> separate ANY]
feature
    g: G
        do
            Result := Void
        end
end

class B [?G -> X]
inherit A [G]          -- Allowed
    -- g is correct here
end

class D [G -> X]
inherit A [G]          -- Prohibited
    -- g is invalid here
end
```

Figure 9.11: Self-initialising formal generic parameters under inheritance

Conformance of generic types

The general conformance rule of Eiffel (8.14.6 /VNCC/ in [53]) stipulates that a generically derived type *B* [*Y*] conforms to *A* [*X*] if and only if *B* conforms to *A*, and *Y* to *X*. The rule seems to be correct: intuitively, if the base classes and all the actual generic parameters conform, then the entire types should conform as well. Unfortunately, the rule is unsound, as illustrated in figure 9.12. If the type of *a2* conforms to the type of *a1*, i.e. $(!, \bullet, A[(!, \bullet, X)]) \preceq (!, \bullet, A[(!, \bullet, X)])$, then it is possible to perform a polymorphic assignment *a1* := *a2*. But now passing the separate entity *my_x* as actual argument to *a1*.*r* results in a catcall because the expected formal argument should be non-separate. Although no inheritance and feature redefinition are involved, the

```
class A [G -> separate X]
feature
    item: G

    r (v: G)
        do
            v.f
        end
end

-- in class C
a1: A [separate X]
a2: A [X]
my_x: separate X
 ...
my_x := a1.item
a1 := a2
a1.r (my_x)        -- Catcall!
```

Figure 9.12: Problems with the covariant conformance rule for generic types

problem is caused by the "redefinition" of the formal argument type: the target has the type $(!, \bullet, A[(!, \bullet, X)])$, so its routine *r* expects an argument conforming to $(!, \bullet, X)$. The actual has a non-conforming type $(!, \top, X)$; hence the catcall. (Redefinition of formal arguments was discussed in detail in section 9.1.) This problem is not concurrency-specific; it is also present in sequential languages that allow covariant subtyping of arrays, e.g. Java and C♯; there, it is dealt with by a run-time check.

Given that the problems are similar, can we apply the same solution? Recall that a formal argument of a routine may be redefined covariantly if and only if it becomes detachable (rule 9.1.2). Transposing it to genericity: an actual generic parameter conforms to another actual generic parameter if and only if they are identical, or the former is detachable. But this is not sufficient because now a detachable parameter may conform to an attached one, which raises another conformance issue, illustrated in figure 9.13. After the polymorphic assignment from *b2* to *b1*, the assignment *my_x* := *b1.item* is invalid because its source is possibly void, hence incompatible with its target. One could claim that this is not dangerous because *X* must be self-initialising anyway (as required by the rule 8.12.11 /VTGD/), so *my_x* may be given a default value if *b1.item* yields **Void**. Nevertheless, this is counterintuitive; the source type $(?, \top, X)$ does not conform to the target $(!, \top, X)$. On the other hand, no such problem arises when both types are detachable, e.g. the assignment *my_y* := *b2.item* is correct: *b2.item* either yields an object of type $(?, \bullet, X)$ or **Void**; both conform to the target type $(?, \top, X)$. Rule 9.2.3 captures the conformance relation on generic types.

Definition 9.2.3 (Conformance of generically derived types) *A generically derived class type* $CT' = B[(\gamma_Y, \alpha_Y, Y)]$ *conforms to* $CT = A[(\gamma_X, \alpha_X, X)]$ *if and only if B conforms to A,* $(\gamma_Y, \alpha_Y, Y) \preceq (\gamma_X, \alpha_X, X)$*, and any of the following conditions holds:*

- *Both actual generic parameters are detachable, i.e.* $\gamma_X = \gamma_Y = ?$.

```
class B [?G]
feature
    item: G
end

-- in class C
b1: B [separate X]    -- X is self-initialising
b2: B [?separate X]
b3: B [?X]
my_x: separate X
my_y: ?X
 ...
b1 := b2
my_x := b1.item       -- Problematic
b2 := b3
my_y := b2.item       -- Correct
```

Figure 9.13: Conformance of detachable and attached actual generic parameters

- *Actual generic parameters are identical, i.e.* $\gamma_Y = \gamma_X$, $\alpha_Y = \alpha_X$, *and* $Y = X$.

If CT' *conforms to* CT, *then the type* $U = (\delta, \beta, CT')$ *conforms to* $T = (\gamma, \alpha, CT)$, *that is* $U \preceq T$, *provided that* δ *conforms to* γ, *and* β *conforms to* α.

Let's apply this rule to a few examples. If *LINKED_LIST* conforms to *LIST*, *Y* conforms to *X*, and both *X* and *Y* are self-initialising (to allow their use in the attached form as well), then

- *LINKED_LIST* [*X*] conforms to *LIST* [*X*]
- *LINKED_LIST* [**separate** *X*] conforms to *LIST* [**separate** *X*]
- *LINKED_LIST* [?*Y*] conforms to *LIST* [?*X*]
- *LINKED_LIST* [?*X*] conforms to *LIST* [?**separate** *X*]
- *LINKED_LIST* [?*Y*] conforms to *LIST* [?**separate** *X*]
- *LINKED_LIST* [?**separate** *Y*] conforms to *LIST* [?**separate** *X*]

  but

- *LINKED_LIST* [*Y*] does not conform to *LIST* [*X*]
- *LINKED_LIST* [*Y*] does not conform to *LIST* [**separate** *X*]
- *LINKED_LIST* [**separate** *Y*] does not conform to *LIST* [**separate** *X*]

The usual subtyping rules apply to types built from generically derived class types, as expressed by the second part of the rule 9.2.3. For example, if *LINKED_LIST* [?*Y*] conforms to *LIST* [?*X*], then:

- *LINKED_LIST* [?*Y*] conforms to ?*LIST* [?*X*]
- *LINKED_LIST* [?*Y*] conforms to **separate** *LIST* [?*X*]
- **separate** *LINKED_LIST* [?*Y*] conforms to **separate** *LIST* [?*X*]
- **separate** *LINKED_LIST* [?*Y*] conforms to ?**separate** *LIST* [?*X*]

but

- ?*LINKED_LIST* [?*Y*] does not conform to *LIST* [?*X*]
- **separate** *LINKED_LIST* [?*Y*] does not conform to *LIST* [?*X*]
- ?**separate** *LINKED_LIST* [?*Y*] does not conform to **separate** *LIST* [?*X*]

Rule 9.2.3 applies to arbitrarily nested generic types, e.g. it is possible to conclude that

LINKED_LIST [?*LINKED_STACK* [?**separate** *LINKED_QUEUE* [?*BUFFER* [?*Y*]]]]

conforms to

separate *LIST* [?**separate** *STACK* [?**separate** *QUEUE* [?*BUFFER* [?**separate** *X*]]]]

provided that *LINKED_LIST* conforms to *LIST*, *LINKED_STACK* to *STACK*, *LINKED_QUEUE* to *QUEUE*, and *Y* to *X*.

9.2.7 Discussion

The genericity mechanism is integrated with SCOOP in a straightforward manner. Most validity and conformance rules need little refinement, with the exception of the generic type conformance rule (8.14.6 /VNCC/) whose part concerning generic types must be replaced by the stronger rule 9.2.3. This rule, however, is not concurrency-specific; it is necessary to close a loophole in the Eiffel type system permitting the occurrence of catcalls due to the covariant conformance of actual generic parameters.

A different approach to the detachability of formal generic parameters has been proposed for Spec♯ by Fähndrich et al. [55]; covariant subtyping of generic types is prohibited but a more flexible use of formal generic parameters is allowed. A formal parameter *G* may be used both as attached and detachable (*nullable* in the Spec♯ jargon) within the same class; it suffices to decorate it with the appropriate tag, e.g.

```
public !G get {...}          // Here, G is non−nullable
public put (?G v) {...}      // Here, G is nullable
```

This enables safe handling and initialisation of arrays with elements of non-nullable types.

In our type system, the locality of actual generic parameters of a type is relative to the instance of the generic class which serves as base class for the type (see sections 9.2.3 and 9.2.4). This approach is similar to Ownership Generic Java (OGJ) by Potanin et al. [124], where the ownership annotations are relative to the generic class whose actual parameter they decorate. For example, a container typed as “my box of books” in OGJ belongs to the current object but the ownership of its elements (books) is not specified. In SCOOP, where we are

concerned with the locality of objects rather than their ownership, this corresponds to a non-separate container of separate books, e.g. *l*: *LIST* [**separate** *BOOK*]. As pointed out in section 5.11, one could take a different approach and evaluate the separate annotations with respect to the client that declares the generic type; in that case, each type annotation would be relative to the current object. For example, the container *l*: *LIST* [**separate** *BOOK*] is non-separate from **Current**; its elements are (potentially) separate from **Current** but we make no assumption about their locality with respect to the container. (Logically, they are also separate from the container, so the class *LIST* must accept separate actual parameters.) On the other hand, the elements of *sl*: **separate** *LIST* [*BOOK*] are non-separate from **Current** but the container itself is. Dietl et al. [48] follow this approach to combine genericity and ownership in their GUT system. This alternative is indeed more convenient for ownership annotations: it facilitates the implementation of common data structures such as collections with iterators, or shared buffers. In the concurrent context, however, passing references across processors' boundaries is much simpler with our approach: only the outermost annotation changes; the rest of the type stays intact, e.g.

LIST [*A* [**separate** *B* [*X*]]]

becomes

separate *LIST* [*A* [**separate** *B* [*X*]]]

With the alternative approach, all actual generic parameters need to be modified due to the viewpoint adaptation: since the processor tags are relative to **Current**, they change when **Current** denotes a different object. This would unnecessarily complicate the language and type rules; some operations may yield invalid types, in particular in the context of constrained genericity. Therefore, we have decided to follow the approach presented in earlier sections.

9.3 Agents

The agent mechanism provides a convenient way to represent partially or completely specified computations as first-class citizens of the object-oriented world. The Eiffel standard [53] gives the following justification for the use of agents:

> In the object-oriented world, objects represent information equipped with operations. These are clearly defined concepts; no one would mistake an operation for an object. For some applications — graphics, numerical computation, iteration, reflection (a system's ability to explore its own properties) — you may find the operations so interesting that you will want to define objects to represent them, and pass these objects around to software elements which can use these objects to execute the operations whenever they want. Because this separates the place of an operation's definitions from the place of its execution, the definition can be incomplete, since you can provide any missing details at the time of any particular execution.

Syntactically, an agent is of the form **agent** *x*.*f* (...) or **agent** *f* (...) with the following possible variants:

- Any argument may be replaced by a question mark '?', making the argument open.

- The target may be replaced by $\{T\}$, where *T* is the name of a type, making the target open.
- The argument list (...) may be removed altogether, making all arguments open.

Agents combine the expressiveness of higher-order functionals with the safety of a static type system. They may be used for a number of purposes but they prove particularly useful in event-driven programming [12, 95]. They enable a loose coupling of software components: an object may call another object's feature without becoming its explicit client. Three principal characteristics of agents turn them into a powerful modelling and implementation tool:

- *Mobility*
  An agent may be created by one object and passed to other objects for a later use.
- *Separation of agent construction and agent execution*
  An agent is constructed once but it may be executed several times, at any convenient time.
- *Open operands*
  Operands required by the computation (including the target object) may be left unspecified (open) until the call-time; a mix of fixed and open operands may be used.

Agents are well integrated with Eiffel although the mechanism is not completely type-safe in its current form (some run-time checks are necessary). Extending it to cover concurrency seems to be straightforward but a closer examination reveals a number of potential difficulties:

- How can we prevent an agent from becoming a traitor?
- Can agents be passed across a processor's boundary?
- How can we ensure the conformance of the actual arguments to the open operands at call-time?
- Can we safely use open-target agents in a concurrent context?

This section sketches a solution. First, we discuss the type safety and propose an agent creation rule which eliminates the danger of traitors. Second, we show how mobility of agents can be achieved without compromising type safety. We also describe the treatment of open arguments. Third, we propose a validity rule for open-target agents. Finally, we present the practical advantages brought to SCOOP by the enriched mechanism.

9.3.1 Agents as potential traitors

An agent represents a feature ready to be called. In fixed-target agents, i.e. those declared with an explicit target, the object to which the feature is applied is known at the time of agent creation; the static type of the agent is derived from the static type of its target and the types of formal arguments corresponding to open operands. Consider an expression **agent** $x.f$ (...) where x has the type T_x. Let $i_1, ..., i_m$ $(m \geq 0)$ be its open operand positions, if any, and let $Ti_1, ..., Ti_m$ be the types of f's formal arguments at positions $i_1, ..., i_m$ (taking Ti_1 to be T_x if $i_1 = 0$). According to the rule 8.27.17 in sequential Eiffel [53], the expression has the type

- *PROCEDURE* [T_x,*TUPLE* [$Ti_1, .., Ti_m$]] if f is a procedure.

- *FUNCTION* [T_x,*TUPLE* [$Ti_1, .., Ti_m$], R] if f is a function of result type *R* other than *BOOLEAN*.
- *PREDICATE* [T_x,*TUPLE* [$Ti_1, .., Ti_m$]] if f is a function of result type *BOOLEAN*.

A naive application of this rule in SCOOP leads to the creation of agent objects which are not separate from their creators, even if they represent a separate call. Consider the following code excerpt:

```
my_x: separate X
my_agent: PROCEDURE [separate X, TUPLE]
 ...
my_agent := agent my_x.f
my_agent.call ([]) -- Traitor
```

Assuming that the feature f in class X does not take any arguments and returns no result, the expression **agent** *my_x.f* has the type $(!, \bullet, PROCEDURE[(!, \top, X), (!, \bullet, TUPLE)]))$; *my_agent* is typed correspondingly. But this means that *my_agent* is non-separate, thus controlled in any context; calls on *my_agent* do not have to be wrapped in an enclosing routine. For example, the call *my_agent.call* ([]) is accepted by the compiler. But this is like calling *my_x.f* without locking *my_x*'s processor! Therefore, *my_agent* is a traitor: it provides an unsynchronised access to a separate object. Figure 9.14 illustrates this scenario: the agent *o3* is handled by the same processor as the client *o1*, so the client expects its call to be non-separate; calling *o3*, however, results in an illegal separate call on *o2*.

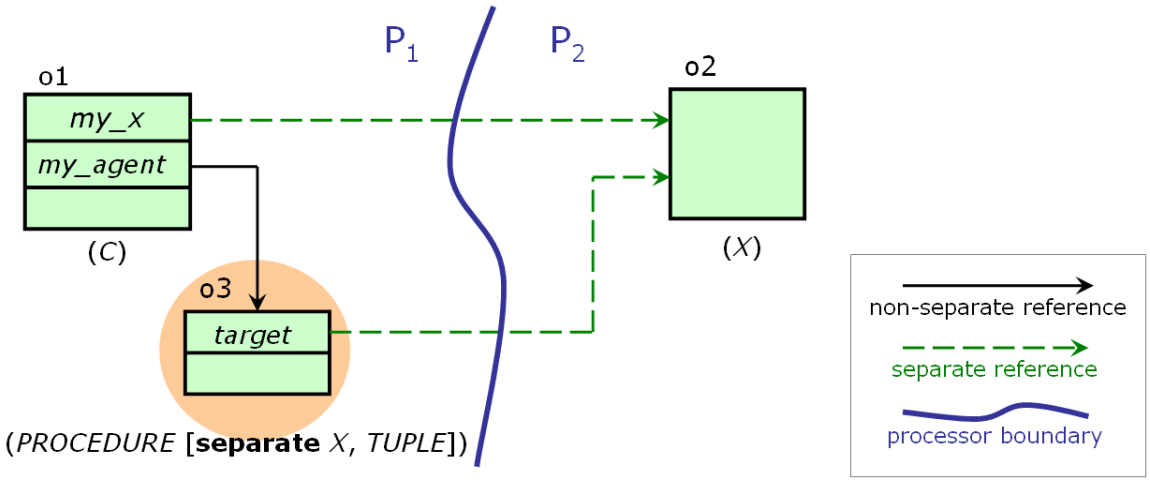


Figure 9.14: Agent as potential traitor

The semantics of agent calls could be changed so that a lock is acquired on the target before applying the feature, e.g. the call *my_agent.call* ([]) is translated into a sequence of operations:

1. Lock the handler of *my_agent*'s target.
2. Perform the asynchronous call represented by *my_agent*.
3. Unlock the target's handler.

This solves the problem of traitors but other issues remain:

- It is difficult to ensure the atomicity of a sequence of agent calls on the same target because the lock on the target's handler is released and acquired again between the subsequent calls.
- If an agent is passed across a processor's boundary then, like any other reference, it becomes separate. To call such an agent, clients first need to lock its handler (which happens to be its creator's handler). This locking serves no real purpose; the client is interested in the target's handler, not the creator's. Figure 9.15 illustrates this point. Assume that *o1* has constructed the agent *o3* and passed it to *o4*. The agent is separate with respect to *o4*, so the latter needs a lock on P_1 before calling *o3*. When *o3* is called by *o4*, it needs to lock P_2 to apply the feature on its target *o2*. Although the client *o4* is handled by the same processor as the agent's target — which means that no locking should be necessary — two lock requests are needed. This may even lead to a deadlock: the lock request issued by the agent blocks until P_2 becomes free, but P_2 may never become free if *o4* decides to execute some query call on *o1*. In that case, P_2 is blocked waiting for P_1 and vice-versa.
- The locality of open operands is assessed with respect to the agent's creator rather than its target. This violates the type safety: an operand that must be non-separate from the target — as in **agent** *my_x.g* (?), provided that *g* is declared as *g* (*y*: *Y*) **do** ... **end** — will appear as separate in the agent. As a result, an actual argument passed to the agent at the call time may be handled by any processor, although the expected formal is non-separate; a traitor may be introduced.

Therefore, changing the semantics of an agent call is not a good solution. A different approach is needed: the creation rule for agents must be refined to capture correctly the locality of their target and operands; agent creators should be clearly decoupled from agent users to avoid the unnecessary locking. The primary concern, however, is the elimination of traitors without modifying the semantics of the agent call.

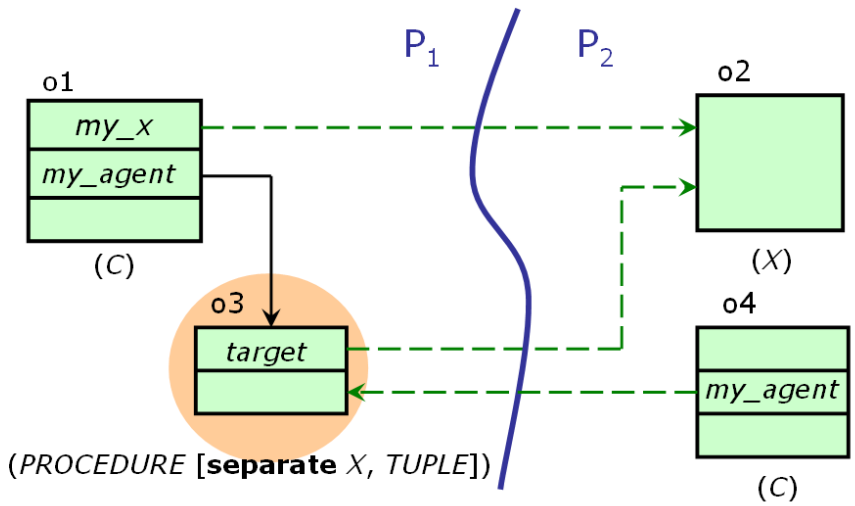


Figure 9.15: Problematic agent

9.3.2 Separate agents

To achieve full decoupling of agent creation and agent execution, we need to ignore the details concerning the object which creates an agent; this object does not intervene in the interaction between the agent and its clients, so its type (more precisely: its locality with respect to the target) should have no impact on the agent's type. On the other hand, the agent has a strong bond with its target; the simplest way to express this relationship is to place the agent on its target's processor, hence make the agent separate from the creator object if necessary, as illustrated in figure 9.16. Rule 9.3.1 captures the new semantics of agent creation.

Definition 9.3.1 (Agent creation rule) *A newly created agent is placed on its target's processor.*

The rule eradicates several problems at once:

- Agents cannot become traitors anymore: an agent call is only possible in a context where the target's processor is locked.
- A sequence of calls on the same agent, appearing in a routine that locks the agent's handler, is guaranteed to execute atomically.
- The agent bears no indication of its creator; on the other hand, it is strongly coupled with its target.
- Open operands are correctly typed because their separateness from the agent implies the separateness from the target; conversely, their non-separateness from the agent implies the non-separateness from the target.

Rule 9.3.2 refines the Eiffel rule 8.27.17 [53] to reflect the new creation semantics.

Definition 9.3.2 (Agent expression type) *Consider an expression* ***agent*** $x.f\ (...)$ *where x has the type* $T_x = (!, \alpha, X)$. *Let* $i_1, ..., i_m$ $(m \geq 0)$ *be its open operand positions, if any, and let* $Ti_1, ..., Ti_m$ *be the types of f's formal arguments at positions* $i_1, ..., i_m$ *(taking* Ti_1 *to be* T_x *if* $i_1 = 0$*). The expression has the type*

- $(!,\ \alpha,\ PROCEDURE\ [\ (!,\ \bullet,\ X),\ (!,\ \bullet,\ TUPLE\ [Ti_1,\ ..,\ Ti_m])\ ]\ )$
  if f is a procedure.
- $(!,\ \alpha,\ FUNCTION\ [\ (!,\ \bullet,\ X),\ (!,\ \bullet,\ TUPLE\ [Ti_1,\ ..,\ Ti_m]),\ T_R\ ]\ )$
  if f is a function of result type T_R *other than* $(!, \bullet, BOOLEAN)$.
- $(!,\ \alpha,\ PREDICATE\ [\ (!,\ \bullet,\ X),\ (!,\ \bullet,\ TUPLE\ [Ti_1,\ ..,\ Ti_m])\ ]\ )$
  if f is a function of result type $(!, \bullet, BOOLEAN)$.

All these types conform to $(!,\ \alpha,\ ROUTINE\ [\ (!,\ \bullet,\ X),\ (!,\ \bullet,\ TUPLE\ [Ti_1,\ ..,\ Ti_m])\ ]\ )$

The fact that an agent is handled by its target's processor is reflected in its type: it has the same processor tag as the target. For example, if the targets *my_x* and *my_y* and the features *f* and *g* are declared as

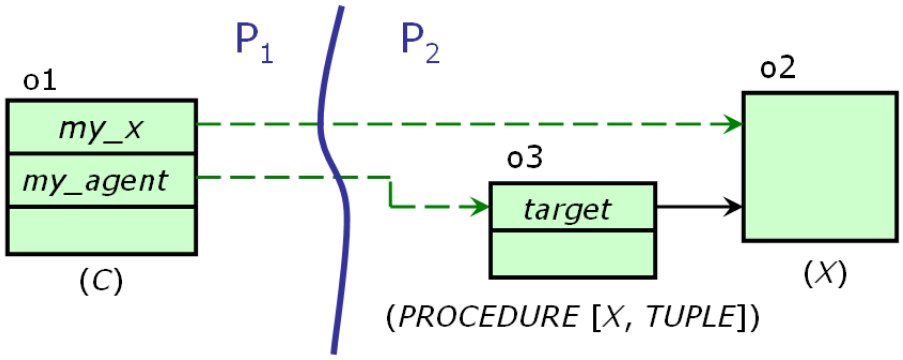


Figure 9.16: Separate agent

```
-- in class C
my_x: separate <px> X
my_y: X
 ...
a := agent my_x.f
b := agent my_y.f

-- in class X
f: ?Y
g: separate Y
```

so that *my_x* has the type $(!, px, X)$ and *my_y* has the type $(!, \bullet, X)$, then the expression **agent** *my_x.f* has the type

$$(!, px, PROCEDURE[(!, \bullet, X), (!, \bullet, TUPLE), (?, \bullet, Y)])$$

whereas the expression **agent** *my_y.f* has the type

$$(!, \bullet, FUNCTION[(!, \bullet, X), (!, \bullet, TUPLE), (!, \top, Y)])$$

The variables *a* and *b* used for storing the agents are declared correspondingly:

```
a: separate <px> FUNCTION [X, TUPLE, ?Y]
b: FUNCTION [X, TUPLE, separate Y]
```

As you can see, the agents and their targets have the same processor tags: *px* for *a* and *my_x*, and '•' for *b* and *my_y*. The target type is always non-separate with respect to the agent; that is why it appears here as $(!, \bullet, X)$. The result type — for agents representing a function — corresponds to the declared result type of the function; it does not matter whether the target is separate or not. Therefore, the result type is $(?, \bullet, Y)$ for *a*, and $(!, \top, Y)$ for *b*. The tuple of operands is always non-separate from the agent; we will come back to this point in section 9.3.2. A direct call *b*. *call* ([]) is valid because *b* is controlled (see rule 6.5.3). On the other hand, a direct call *a*. *call* ([]) is now invalid because it happens in a context where *a* is not controlled. The call must be wrapped in a routine which locks *a*'s handler, e.g. by taking *a* as actual argument:

```
call (an_agent: separate ROUTINE [ANY, TUPLE])
        -- Execute operation represented by an_agent.
```

```
    do
        an_agent . call ([])
    end
...
call (a)
```

The call *an_agent . call ([])* in the body of *call* is valid; it has the same effect as *a . call ([])* or *my_x.f* but without violating the synchronisation policy. (As a matter of fact, feature *call* may be used for wrapping arbitrary agent calls without operands; see section 9.3.4.) But it only executes a single call; if an atomic sequence of agent calls is needed, an enclosing routine has to be written for that purpose, e.g.

```
call_three_times (an_agent: separate ROUTINE [ANY, TUPLE])
        -- Execute operation represented by an_agent .
    do
        an_agent . call ([])
        an_agent . call ([])
        an_agent . call ([])
    end
...
call_three_times (a)
```

Agents behave now just like any other entity; the same validity rules apply. Therefore, agent calls may be freely mixed with other feature calls; references to agents may be passed across processors' boundaries, etc. Additionally, the semantics of agent calls is exactly the same as that of usual calls: calling an agent amounts to requesting the application of one of its features; no hidden synchronisation occurs.

Agent mobility

When a reference to an agent is passed across the boundary of a processor (in a separate feature call), its type changes due to the type combination rules (see section 6.4). Typically, the reference is seen as separate on the receiver's side. With the naive application of the sequential agent semantics, it would cause the problem depicted in figure 9.15: unnecessary locking and potential deadlock. No such problems occur with the new semantics captured by the rule 9.3.1; since the agent is non-separate from its target, the reference becomes separate if and only if the target does. Consider the following code excerpt:

```
-- in class C
my_agent: separate PROCEDURE [X, TUPLE]
pass_my_agent ( receiver : separate C)
    do
        receiver . set_my_agent (my_agent)
    end
```

Figure 9.17 illustrates this situation. The current object *o1* holds a reference (*my_agent*) to *o3*, and passes it to another object *o4* by calling *receiver . set_my_agent (my_agent)*. The receiver *o4* sees the reference to *o3* as separate, hence no traitor is introduced: *o4* may only perform a call on *o3* (thus, implicitly, on *o2*) in a context where P_2 is locked. In this particular example,

the agent was already separate before being passed across the processor's boundary. But even a non-separate agent becomes separate when passed to a separate object, just like a non-separate reference would. On the other hand, passing an agent to a non-separate object preserves its processor tag. Through the combination of rule 9.3.1 with the rules **T-QFCallQual** and **T-CCallQual** (see section 6.11), the locality of the target is always reflected in the locality of the agent; therefore, the type safety is preserved.

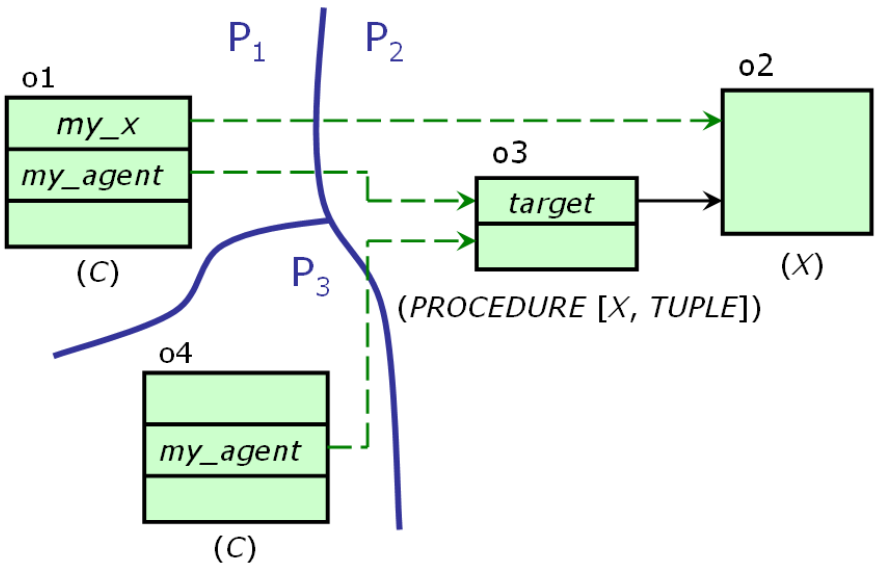


Figure 9.17: Agent mobility

Result types

Agents may represent commands or queries; the latter have a result that can be retrieved after an agent call, using the feature *last_result*, e.g.

```
my_agent.call ([])
...
res := my_agent.last_result
```

To ensure the type safety, a result must not be retrieved before at least one agent call has been performed; the precondition of *last_result* enforces it. The type of the result corresponds to the type of the agent combined with the result type of the routine represented by it. (A direct call to the feature represented by *my_agent* yields the same type, which is precisely what we want.) The type combinator '$\star$' is used for calculating the actual type, just like for non-agent query calls (see the rule **T-QCallQual**). For example, if *my_agent* is defined as

my_agent: **separate** *FUNCTION* [*X*, *TUPLE*, ?*Y*]

which means that it represents a separate call to a query, say *my_x.g*, where *g* returns a result of type $(?, \bullet, Y)$, then *my_agent*. *last_result* has the type $(?, \top, Y)$ because

$$(!, \top, FUNCTION[...]) \star (?, \bullet, Y) = (?, \top, Y)$$

in which case the target of the above assignment has to be declared as

res: ?**separate** *Y*

If *my_agent* is non-writable, e.g. if it is a formal argument, then we can take advantage of the *implicit type* rule 6.2.3 to give it a more precise type $(!, my\_agent.\mathbf{handler}, FUNCTION[...])$; the result type will also be more precise:

$$(!, my\_agent.\mathbf{handler}, FUNCTION[...]) \star (?, \bullet, Y) = (?, my\_agent.\mathbf{handler}, Y)$$

Therefore, *res* may be declared as

res: ?**separate** <*my_agent*.**handler**> *Y*

or, taking advantage of polymorphism, as

res: ?**separate** *Y*

Open operands

One of the useful properties of Eiffel's agent mechanism is the possibility to leave open certain arguments of the wrapped feature; these arguments have to be supplied at the call time. For example, if a feature *g* in classes *X* and *Y* is declared as

g (*a*: ?*A*; *b*: **separate** *B*) **do** ... **end**

then an agent may be declared without fixing the arguments, e.g.

my_x: **separate** *X*
my_agent_two_open := **agent** *my_x.g* (?, ?)

or with just one fixed argument, e.g.

my_y: **separate** <*py*> *Y*
my_agent_one_open := **agent** *my_y.g* (?, *b*)

So far, we have considered agents without open operands. Such agents do not take any actual arguments at the call time; strictly speaking, there is one actual argument — an empty tuple — but no "real" arguments. The issue of type safety becomes more complex in the presence of open operands. We have to make sure that no traitors are introduced as a result of argument passing at the call time; this is more difficult to capture statically than the separateness of the target or the result type of an agent. In the above example, the routine *g* expects one argument of type $(?, \bullet, A)$ and one of type $(!, \top, B)$. Assuming that the call

my_agent_two_open.call ([*my_a*, *my_b*])

happens in a context where *my_agent_two_open* is controlled, what types should the actual arguments *my_a* and *my_b* have? Ignoring for the moment the tuple wrapping both arguments, rule **T-CCallQual** requires that they conform to the type of *my_agent_two_open* combined (using the operator '$\otimes$') with the type of the corresponding formal arguments. Therefore, *my_a* should conform to $(?, \bot, A)$ because $(!, \top, X) \otimes (?, \bullet, A) = (?, \bot, A)$, whereas *my_b* should conform to $(!, \top, B)$ because $(!, \top, X) \otimes (!, \top, B) = (!, \top, B)$. The latter may be declared as

my_b: **separate** *X*

or

my_b: **separate** <*p*> *X*

or even

my_b: *X*

because all these declarations ensure the type conformance. On the other hand, *my_a* has to conform to $(?, \bot, A)$; but only **Void** conforms to this type! So the call must appear as

my_agent_two_open.call ([**Void**, *my_b*])

Does it mean that separate agents expecting non-separate operands are useless? Not at all; we have already observed a similar problem with actual arguments of separate calls in section 6.4, and we know how to deal with it. Qualified and unqualified processor tags may be used to capture the relative non-separateness of the agent and the actual argument, resulting in a more precise type combination. For example, if *my_agent_two_open* was an attached formal argument of the enclosing routine, then it would suffice to declare *my_a* as

my_a: **separate** <*my_agent_two_open*.**handler**> *A*

Thanks to the *implicit type* rule 6.2.3, *my_agent_two_open* has the processor tag $my\_agent\_to\_open.\mathbf{handler}$; as a result of the type combination *my_a* needs to conform to $(?, my\_agent\_two\_open.\mathbf{handler}, A)$. The problem may also be solved using an unqualified processor tag. Consider the agent *my_agent_one_open* declared above: it has the processor tag py (because of its target); therefore, the call

my_agent_one_open.call ([*my_a*])

expects an argument of type $(?, py, A)$ because $(!, py, X) \otimes (?, \bullet, A) = (?, py, A)$. So, *my_a* simply has to be declared as

my_a: **separate** <*py*> *A*

The type combinator '$\otimes$' is very convenient here: the types of the actual arguments in an agent call are calculated in a similar way as for other calls. The only difference is that the tuples wrapping actual and formal arguments must be transparent to the type checker. Therefore, we enrich the type system with the rules **T-ACall** and **T-ASetOperands** which strip down the actual and the formal arguments to check their conformance. The additional rules are used to type-check calls to features *call* and *set_operands* of class *ROUTINE* and its descendants. Other feature calls are taken care of by the rules given in chapter 6.

$$\frac{\begin{array}{l}\Gamma \vdash e : T_e,\ \ \Gamma \vdash isControlled(T_e),\ \ \Gamma \vdash ClassType(T_e) = ROUTINE\ [T_t, T_{arg}] \\ \Gamma \vdash T_{arg} = (!, \bullet, TUPLE\ [T_1, ..., T_n])\ \ \ n \geq 0 \\ \Gamma \vdash a : (!, \bullet, TUPLE\ [T'_1, ..., T'_n]),\ \ \Gamma \vdash \forall_{i \in 1..n}\ \ T'_i\ \preceq T_e \otimes T_i\end{array}}{\Gamma \vdash e.call\ (a)\ \diamond}$$

(T-ACall)

$$\frac{\begin{array}{l}\Gamma \vdash e : T_e,\ \ \Gamma \vdash isControlled(T_e),\ \ \Gamma \vdash ClassType(T_e) = ROUTINE\ [T_t, T_{arg}] \\ \Gamma \vdash T_{arg} = (!, \bullet, TUPLE\ [T_1, ..., T_n])\ \ \ n \geq 0 \\ \Gamma \vdash a : (!, \bullet, TUPLE\ [T'_1, ..., T'_n]),\ \ \Gamma \vdash \forall_{i \in 1..n}\ \ T'_i\ \preceq T_e \otimes T_i\end{array}}{\Gamma \vdash e.set\_operands\ (a)\ \diamond}$$

(T-ASetOperands)

9.3.3 Open targets

Rule 9.3.2 works correctly for fixed-target agents, i.e. agents whose target is supplied at the construction time. On the other hand, the actual target to which the feature represented by an open-target agent applies is only known at the call time; all we know at the construction time is the type to which the target must conform. But this may not be enough, as illustrated in the following code excerpt:

```
a: PROCEDURE [X, TUPLE [X]]
b: separate PROCEDURE [X, TUPLE [separate X]]
...
a := agent {X}.f
b := agent {separate X}.f
```

Performing a call on the non-separate agent a is not a problem because a is controlled, so we may simply write $a.call\ ([x])$, where x must be of type $(!, \bullet, X)$. But there is no way to perform a call on b. Since b is separate, a direct call $b.call\ ([my\_x])$ is invalid, even if my_x has the required type $(!, \top, X)$; it is necessary to wrap the call into a routine that takes b as actual argument, e.g.

```
call_open_target (an_agent: separate PROCEDURE [X, TUPLE [separate X]])
    do
        an_agent.call ([my_x])
    end
...
call_open_target (b)
```

The feature application rule 6.1.5 requires *call_open_target* to lock the processor that handles its formal argument. But what processor should be locked? Since b's target has not been fixed at construction time, b's processor is unknown; the call *call_open_target* (b) cannot be executed correctly. We cannot postpone the synchronisation step until $an\_agent.call\ ([my\_x])$ because this would alter the semantics of the agent call, bringing back all the problems discussed at the beginning of this section. Therefore, we prohibit processor tags other than '•' in the open target type specification, i.e. an open-target agent may only be of the form

```
agent {X}.f (...)
```

but not

```
agent {separate X}.f (...)
```

or

```
agent {separate <p> X}.f (...)
```

The use of detachable types for open targets, as in **agent** $\{?X\}.f$, leads to similar problems; therefore, such types are prohibited as well. Rule 9.3.3 summarises the restrictions on the open-target agents.

Definition 9.3.3 (Open-target agent construction) *An open target agent must have an attached and non-separate target type, i.e. it must be of the form* ***agent*** $\{X\}.f\ (...)$.

The restriction imposed by the above rule does not limit the usefulness of the mechanism. Calling an open-target agent on a separate target is possible but requires a bit more machinery: an "envoy" object must be placed on the target's processor; its task consists of importing the agent and calling it with the target as argument. Since the envoy is non-separate from the target and the agent (the agent has been imported), the call is non-separate. Section 10.2.3 demonstrates how this technique is used to implement resource pooling and load balancing.

Discussion

The rules introduced in this section permit the use of agents in a concurrent context. Our solution handles agents with and without open operands, fixed-target and open-target agents, and inline agents [53]. The modifications of the standard rules have been kept to a bare minimum; most importantly, the effect of executing an agent conforms to the standard semantics defined in the Eiffel standard, and captured by the rule 9.3.4 below.

Definition 9.3.4 (Effect of agent execution) *Let D0 be an agent object with associated feature f and open operand positions $i_1, ..., i_m (m = 0)$. The information in D0 enables a call to the procedure call, executed at any call time posterior to D0's construction time, with target D0 and (if required) actual arguments $ai_1, ..., ai_m$, to perform the following:*

- *Produce the same effect as a call to f, using the closed operands at the closed operand positions and $ai_1, ..., ai_m$, evaluated at call time, at the open operand positions.*
- *In addition, if f is a function, setting the value of the query last_result for D0 to the result returned by such a call.*

9.3.4 Applications of separate agents

Besides increasing the expressiveness of our framework by providing a convenient way to represent asynchronous computations, agents bring a number of immediate benefits to the programmer. We discuss here two practical problems identified in chapter 5 — burdensome enclosing routines and the lack of full asynchrony — and show how separate agents solve them. Further benefits of the agent mechanism, e.g. the possibility to "wait faster", the *rendezvous*-style synchronisation, and event-driven programming, are described in chapter 10.

Convenience: universal enclosing routine

SCOOP prohibits calls on uncontrolled expressions (see section 6.5); this forces programmers to provide an enclosing routine to wrap each call on a separate target. This practice is justified if we want to specify some precondition or perform an atomic sequence of calls on the same target. On the other hand, it may become tedious if we only want to perform a single separate call and need no precondition. We are still forced to write an additional routine for each call, just for the sake of proper synchronisation via argument passing, as illustrated in figure 9.18.

The agent mechanism comes in handy: the simple feature

```
my_x: separate X
 ...
r (my_x)
s (my_x, ''Hello world!'')
t (my_x, 5)
 ...
r (x: separate X)
    do
        x.f
    end

s (x: separate X; s: STRING)
    do
        x.g (s)
    end

t (x: separate X; i: INTEGER)
    do
        x.h (i)
    end
```

Figure 9.18: Burdensome enclosing routines

```
my_x: separate X
 ...
call (agent my_x.f)
call (agent my_x.g (''Hello world!'') )
call (agent my_x.h (5) )
 ...
call ( a_feature : separate ROUTINE [ANY, TUPLE])
        -- Universal enclosing routine .
    do
        a_feature . call ([])
    end
```

Figure 9.19: Universal enclosing routine

```
call ( a_feature : separate ROUTINE [ANY, TUPLE])
        -- Universal enclosing routine .
    do
        a_feature . call ([])
    end
```

acts as universal enclosing routine providing the necessary synchronisation. Figure 9.19 shows how our previous example can be rewritten in a clear and compact fashion. Additional enclosing routines disappear; each separate call is simply wrapped in an agent and passed as actual argument to *call*. One could argue that writing *call* (**agent** *my_x.f*) puts less burden on the

programmer than defining a custom enclosing routine but it is still more verbose than *my_x.f*. We see this little syntactic burden as an advantage: it makes the semantic difference visible. A sequence of calls

```
call (agent my_x.f)
call (agent my_x.g ('' Hello world !'') )
call (agent my_x.h (5) )
```

will never be mistaken for an atomic sequence; on the other hand, a sequence

```
my_x.f
my_x.g ('' Hello world !'')
my_x.h (5)
```

could.

Expressiveness: full asynchrony

As pointed out in section 5.9, full asynchrony is impossible to achieve in SCOOP_97 because separate calls are wrapped in enclosing routines; calls to these routines are potentially blocking. So, at least one synchronisation point is required before performing an asynchronous call; this means that separate calls are, in reality, *quasi-asynchronous*. Figure 9.20 illustrates a typical scenario: although *l . write* (...) and *m.send* (...) are procedure calls (i.e. they are asynchronous), the client blocks when executing *log_event* and *send_message* until the necessary lock is acquired; this introduces synchrony. Obviously, the client would prefer not to wait at all, in particular for activities such as logging and e-mail communication; it would be much more convenient for the client to continue its activities knowing that these calls will be taken care of at some point in the future. The client does not care about the precise timing but it may be interested in preserving the ordering of calls on the same target, i.e. the event "Switzerland vs. France 0:0" should be logged before the event "Germany vs. Poland 2:0".

Agents provide a way to solve this problem in SCOOP: a separate call may be wrapped in an agent and passed to a routine which takes care of its asynchronous execution, as illustrated in figure 9.21. Routine *asynch* takes as formal argument an agent representing a separate call, creates a separate executor object to handle that call, and launches the executor. Since the formal argument of *asynch* is detachable and because the executor is handled by a freshly created processor, which is different from the processors handling the client and the agent, feature *asynch* is non-blocking. The client immediately proceeds with the execution of the next instruction; the executor will take care of calling the requested feature as soon as it can get hold of the agent's processor. Figure 9.22 illustrates the object structure resulting from the execution of

```
asynch (agent log . write ('' Switzerland vs. France 0:0'') )
```

(Objects depicted in yellow are created implicitly by the mechanism; they are "transparent" to the client.) Although the client can make no assumptions about the time elapsed before the actual call is performed, the mechanism preserves the FIFO ordering of calls on the same processor; asynchronous calls on different processors may execute in any order. In our example, the call

```
log . write ('' Switzerland vs. France 0:0'')
```

is guaranteed to execute before

```
log: separate LOG
mailer: separate MAILER
...
log_event (log, ''Switzerland vs. France 0:0'')
send_message (mailer, ''all@inf.ethz.ch'', ''Hopp Schwiiz!'')
log_event (log, ''Germany vs. Poland 2:0'')
...
log_event (l: separate LOG; s: STRING)
    do
        l.write (s)
    end

send_message (m: separate MAILER; address, message: STRING)
    do
        m.send (address, message)
    end
```

Figure 9.20: Partial asynchrony

```
asynch (agent log.write (''Switzerland vs. France 0:0'') )
asynch (agent mailer.send (''all@inf.ethz.ch'', ''Hopp Schwiiz!'') )
asynch (agent log.write (''Germany vs. Poland 2:0'') )
...
asynch ( a_feature : ?separate ROUTINE [ANY, TUPLE])
        -- Call a_feature asynchronously.
    local
        executor: separate EXECUTOR
    do
        create executor.execute ( a_feature )
    end
```

Figure 9.21: Full asynchrony with agents

```
log.write (''Germany vs. Poland 2:0'')
```

but the call

```
mailer.send (''all@inf.ethz.ch'', ''Hopp Schwiiz!'')
```

may execute before the other calls, between them, after them, or at the same time (overlapping).

The run-time overhead incurred by the mechanism corresponds to creating a fresh processor and two objects: an executor and an agent object. Requiring a new processor for each asynchronous call may be prohibitively expensive; a smart implementation could reuse processors that have been created for other asynchronous calls and have already finished their job. In that case, the overhead is minimal: two object creations per asynchronous call.

Note that *asynch* is not a new keyword but a library routine provided as part of the *CONCURRENCY* library; so is the class *EXECUTOR* (see section 11.4 and appendix A). Hav-

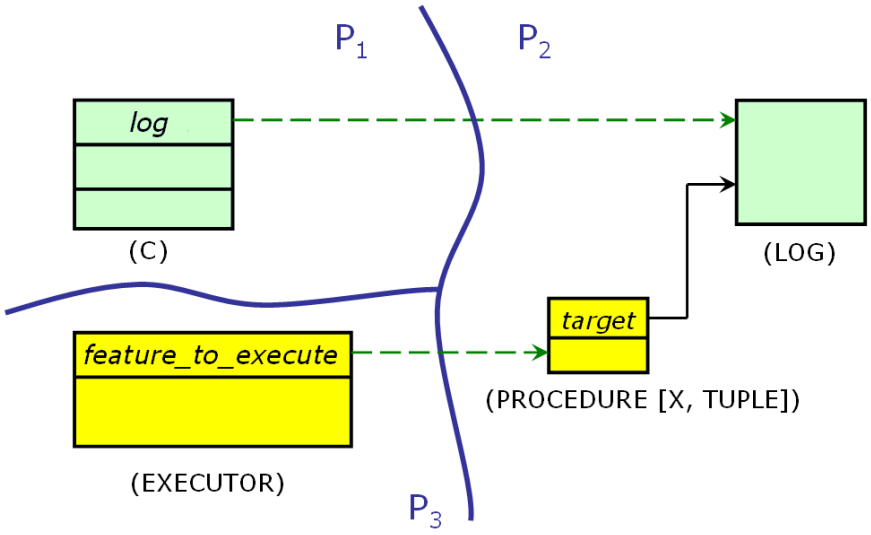


Figure 9.22: Asynchronous executor

ing implemented this useful mechanism, why not notify our PhD supervisor:

asynch (**agent** *mailer*.*send* (''*bm@supervisor.org*'', ''*Full asynchrony solved* .''))

and, knowing his dislike for new keywords, immediately follow with another message:

asynch (**agent** *mailer*.*send* (''*bm@supervisor.org*'', ''*No additional keywords* .''))

Now, we can go back to work (since no waiting is required) knowing that he will eventually receive the messages in the correct order.

Asynchronous self-calls

Since a fully synchronous call decouples completely the client and the supplier objects, i.e. the availability or non-availability of the latter does not influence the actions of the former, nothing prevents us from using this mechanism with non-separate calls, e.g.

```
my_x: X
 ...
asynch (agent my_x.f (...) )  -- Non-separate call
asynch (agent g (...) )       -- Self-call
 ...
```

The calls to *my_x.f* (...) and *g* (...) will not be executed until the target handler — which happens to be the current processor — becomes idle. Of course, it will only become idle after finishing its current activity and being unlocked by its current holder. At that point, the executor object taking care of *my_x.f* will have a chance to lock the current processor and perform the call. When it terminates, another executor taking care of *g* (...) will take its turn and perform the call. So, in a way, the current handler will receive its own calls "from the past".

One may question the utility of this mechanism; in fact, asynchronous self-calls find surprisingly many practical applications. Most importantly, they let a processor sustain its long-term activity: even if it has to become idle for some time, a self-call performed before going idle

will eventually "wake it up" and resume its activity; the rendezvous synchronisation pattern and active objects may be simulated using this mechanism. Self-calls are also useful in event-driven programming. All these topics are discussed in section 10.1.

9.4 Once routines

In Eiffel, a routine *r* may have a *once form* rather than a usual body; it is then shared by all the instances (direct or indirect) of the class where it appears, and executed at most once. Once routines may be procedures or functions. The latter are more common; they may be seen as constants evaluated lazily on first access. Figure 9.23 illustrates their typical use: *stock_statistics* represents a long-lasting computation that should be avoided unless necessary; *shared_printer* returns a reference to a global print manager. Both features should be evaluated at most once.

```
-- in class C
 stock_statistics : STOCK_STATISTICS
        -- Stock statistics since quoting started
    once
        create Result.make_since_beginning_until (date.now)
        -- Long-lasting computation
    end

print_manager: separate PRINT_MANAGER
        -- Manager for shared network printers
    once
        create Result.make (fresh_ip )
    end
 ...
 if not price.is_set then
    price.set ( stock_statistics .average)          -- Lazy evaluation
end
 ...
 print (print_manager, my_document)
 print (print_manager, stock_statistics .as_text ) -- No second evaluation
```

Figure 9.23: Once functions

Once procedures do not raise any particular problems in the context of SCOOP; the usual validity and semantic rules apply, e.g. a once procedure taking an attached argument must lock the corresponding handler when called. Once functions, however, give rise to an interesting problem: should they be evaluated once for all the instances of a given class in the whole system, or just within one processor? Compton [43] suggests the once per processor semantics as the only sound solution; Adrian [2] dismisses this approach and proposes the once per system semantics. The examples in figure 9.23 show that both authors are partly right and partly wrong. Function *stock_statistics* cannot have a once per system semantics because it would introduce a traitor: instances of *C* handled by different processors would share a non-separate reference. The once per system semantics is correct here. On the other hand, *shared_printer* cannot have

a once per processor semantics; the type safety would not be violated but we would end up with as many print manager objects as there are processors in the system. This is clearly not what we want; we need a single print manager shared by all the instances of *C*. Therefore, the once per system semantics must apply.

We apply the once per processor semantics to non-separate functions, and the once per system semantics to separate ones. This preserves the type safety (unlike Adrian's approach) while being more expressive than Compton's solution. Rule 9.4.1 captures the proposed semantics.

Definition 9.4.1 (Semantics of once routines) *Once functions of a separate result type have the* once per system *semantics; their result is shared by all the instances of the declaring class, no matter what processors they are handled by. Once functions of a non-separate result type have the* once per processor *semantics; their result is shared by all instances of the class handled by the same processor. Once procedures have the* once per processor *semantics.*

Mobility issues

Rule 9.4.1 gives an unambiguous semantics to once functions but the *object import* mechanism proposed in section 6.8 (and used implicitly on expanded objects passed across processors' boundaries) complicates the situation a bit. Recall that $x.import$ yields a non-separate copy of the object structure reachable from x via non-separate references and expanded attributes. (The other two versions of the operation — *deep_import* and *flat_import* — behave similarly; the differences are irrelevant here, so we focus on the basic form *import*.) What happens if an imported object is based on a class *C* listing a non-separate once function *f*? According to the above rule, *f* should be evaluated at most once within a given processor, and all the instances of *C* handled by that processor should share *f*'s result. But now a copy of the object is placed on a different processor, where some instance of *C* might have already called *f* and given it a different value. To preserve the semantics of once functions, the semantics of object import must be refined so that the value of *f* in the imported object be set to the value used by other objects on the target processor. If *f* has not yet been evaluated on the target processor, then the value of *f* carried by the imported object becomes the valid value for all the instances of *C* on the target processor.

9.5 Discussion

Unlike most concurrency models claiming a full integration with object technology, SCOOP goes beyond a restricted subset of an O-O language; we provide a support for all the advanced mechanisms needed to use the object technology in practice. It is surprising how few modifications and refinements of the existing language rules are necessary; all we have done here is trying to understand these mechanisms and their interplay with the enriched type system and the generalised semantics of feature call, feature application, argument passing, and assertions, proposed in the previous chapters. The resulting compatibility of the model with the advanced mechanisms demonstrates that, contrary to the (unfortunately still) popular opinion, concurrency and object-orientation are not incompatible and disjoint worlds, but rather the necessary components of the same programming paradigm.

10

Using SCOOP in practice

SCOOP has been applied to solving many practical problems, ranging from typical synchronisation scenarios (producer-consumer, dining philosophers, Santa Claus) to challenging issues of resource pooling, parallel wait on several inputs, emulation of rendezvous-style synchronisation and active objects, to control systems for a physical model of a double-shaft elevator and an arm robot. This chapter shows how solutions to these concurrency problems can be implemented in our framework. Code reuse is also discussed.

Several people contributed to the examples presented here. Erol Koç implemented a number of GUI applications: dining philosophers, producers-consumers (section 10.1), matrix multiplication. Erwin Betschart wrote an EiffelVision GUI for the elevator example (section 10.3.1). Matthias Humbert built the physical elevator and designed its control software (section 10.3.1). Volkan Arslan supervised the above student projects; he also co-authored the elevator application for .NET (section 10.3.1) and implemented the original EVENT_TYPE library [12]. Bertrand Meyer proposed the initial solutions of dining philosophers and elevator in the OOSC2 book [94]. Ganesh Ramanathan built the arm robot and implemented its control software (section 10.3.2).

10.1 Classic examples

The classic synchronisation scenarios presented here are an obligatory part of any discussion on concurrency. We have already used two of them — dining philosophers and producers-consumers — to illustrate the synchronisation mechanism of SCOOP in previous chapters. The OOSC2 book [94] described some of these examples; however, they were just paper designs, whereas the examples described here are fully implemented; some of them have been equipped with a GUI.

10.1.1 Dining philosophers: atomic locking of multiple resources

Dijkstra's dining philosophers scenario [50] illustrates the behaviour of multiple processes acquiring and releasing shared resources. A number of philosophers (5 in the original problem) around a table spend their time eating, thinking, then eating again and so on. To eat, each of them needs two forks, placed to his left and to his right; each fork is accessible to two philosophers. The main problem is to synchronise the access to the forks so that no deadlock occurs and no philosopher is starved.

The solution depicted in figures 10.1 – 10.3 is derived from the one proposed in

```
class PHILOSOPHER
inherit
    GENERAL_PHILOSOPHER
    PROCESS
        rename setup as sit_down undefine sit_down end
create
    make
feature {NONE} -- Implementation
    step
            -- Perform basic tasks.
        do
            think
            eat (left_fork, right_fork)
        end

    eat (l, r: separate FORK)
            -- Eat, having grabbed l and r.
        do ... end
end
```

Figure 10.1: Dining philosopher

```
deferred class GENERAL_PHILOSOPHER
feature -- Initialization
    make (l, r: separate FORK)
        do
            left_fork := l
            right_fork := r
        ensure
            left_fork = l and right_fork = r
        end

feature {NONE} -- Implementation
    think do ... end    -- Think

    sit_down do ... end -- Take your place at the table

    left_fork, right_fork: separate FORK
        -- Forks used for eating
end
```

Figure 10.2: General philosopher

OOSC2. Unlike its predecessor, it does not use separate classes (because they are now prohibited in SCOOP). Class *PHILOSOPHER* inherits from two deferred classes: *GENERAL_PHILOSOPHER* and *PROCESS*. The former implements basic features of philoso-

```
deferred class PROCESS
feature -- Basic operations
    live
            -- Main activity
        do
            from setup until over loop step end
            wrapup
        end

feature {NONE} -- Implementation
    setup    -- Prepare to execute process operations (default: do nothing).
        do end

    wrapup  -- Execute termination operations (default: do nothing).
        do end

    step     -- Execute basic process operations.
        deferred
        end
end
```

Figure 10.3: Process

phers, e.g. thinking. (It has no deferred features but it is marked as deferred to prevent creation of any direct instances.) The latter provides a general implementation of processes; its feature *live* is a (possibly endless) loop performing repetitive actions specified by feature *step*. Since *step* is deferred, it has to be effected in the descendant class *PHILOSOPHER*. One may observe the convenient use of multiple inheritance and feature merging: through the renaming of *setup* as *sit_down* and it subsequent undefinition, the inherited features {*PROCESS*}.*setup* and {*GENERAL_PHILOSOPHER*}.*sit_down* are merged and replace *setup* in the body of *live*.

The remarkable simplicity of the solution — compared to the algorithms found in the literature [51, 81, 41] — is due to SCOOP's ability to perform atomic locking of several processors through a single call. The entire synchronisation is expressed in the call

```
eat (left_fork, right_fork)
```

No additional synchronisation between philosophers, or between a philosopher and his forks, is necessary. In fact, the class *FORK* needs no particular features; it is simply declared as

```
class FORK end
```

The implementation described here does not deadlock. It is also guaranteed to be fair. Even if some philosophers conspire to starve the others, the fair scheduling policy of SCOOP (see section 6.1) ensures that each satisfiable request is eventually serviced. (The actual SCOOPLI scheduler provides an even stronger guarantee: requests are serviced in a FIFO order, i.e. a satisfiable request is never overtaken by another one; see the details in section 11.2.) Compared to the typical algorithms for multithreading and active objects found the literature — necessitating explicit asymmetry in handling philosophers, e.g. by only admitting some philosophers into the

room, or by requiring one philosopher to grab forks in the inverse order — SCOOP offers a simpler way to solve the problem.

A version of this example equipped with a GUI has been implemented by Erol Koç as part of his semester thesis [77]. The implementation uses the EiffelVision 2 library to provide the GUI facilities. Since the *scoop2scoopli* tool was not available at that time, SCOOPLI had to be used directly in the source code, i.e. application classes had to inherit explicitly from the library classes *SCOOP_SEPARATE_CLIENT* and *SCOOP_SEPARATE_SUPPLIER* to implement the required synchronisation pattern. The source code and the executable can be found on the SCOOP project page

`http://se.ethz.ch/research/scoop`

```
class PRODUCER [G] inherit PROCESS
create
    make
feature {NONE} -- Initialization
    make (a_buffer : separate BOUNDED_QUEUE [G])
        do
            buffer := a_buffer
        ensure
            buffer = a_buffer
        end
feature -- Basic operations
    store (a_buffer : separate BOUNDED_QUEUE [G]; e: G)
            -- Store e in a_buffer .
        require
            not a_buffer . is_full
        do
            a_buffer .put (e)
        ensure
            a_buffer .count = old a_buffer .count + 1
        end

    step
            -- Produce element and store it in buffer .
        local
            e: G
        do
            e := ...
            store (buffer , e)
        end
feature {NONE} -- Implementation
    buffer : separate BOUNDED_QUEUE [INTEGER]
end
```

Figure 10.4: Producer

```
class CONSUMER [G] inherit PROCESS
create
    make
feature {NONE} -- Initialization
    make (a_buffer : separate BOUNDED_QUEUE [G])
        do
            buffer := a_buffer
        ensure
            buffer = a_buffer
        end
feature -- Basic operations
    retrieved (a_buffer : separate BOUNDED_QUEUE [G]): G
            -- Element retrieved from a_buffer
        require
            not a_buffer.is_empty
        do
            Result := a_buffer.item
            a_buffer.remove
        ensure
            a_buffer.count = old a_buffer.count - 1
        end

    step
            -- Consume element from buffer.
        local
            e: G
        do
            e := retrieved (buffer)
            ...
        end
feature {NONE} -- Implementation
    buffer : separate BOUNDED_QUEUE [INTEGER]
end
```

Figure 10.5: Consumer

10.1.2 Producers-consumers: condition sychronisation

In the producers-consumers scenario, a number of client objects access a shared buffer; clients are either producers storing elements in the buffer, or consumers retrieving elements from the buffer. We used this scenario in earlier chapters to illustrate various mechanisms of SCOOP, in particular the condition synchronisation and the use of wrapper routines to enclose separate calls; however, no full-fledged implementation has been discussed yet. Figures 10.4 and 10.5 show the implementation of producers and consumers respectively. A few interesting points may be highlighted.

- In the examples appearing in previous chapters, a custom-made class *BUFFER* [*G*]

was used to implement a shared buffer. We take a different approach here: to illustrate the support for code reuse offered by SCOOP, we use an existing library class *BOUNDED_QUEUE* [*G*] (see section 10.4 for a detailed discussion of sequential-to-concurrent code reuse). Its features *put*, *item*, and *remove* provide the required FIFO buffering facilities.

- Following the new semantics of contracts discussed in chapter 8, the postconditions of {*PRODUCER*}.*store* and {*CONSUMER*}.*retrieved* are evaluated asynchronously. This is particularly important in *store* because the call *a_buffer* .*put* (*e*) is asynchronous and producers do not want to wait until this operation has terminated. The producer-consumer scenario in SCOOP_97 [94] excluded these important postconditions to avoid wait by necessity. In SCOOP, no such restrictions apply; the use of postconditions does not reduce the amount of parallelism.

- At the very abstract level, producers and consumers are similar to dining philosophers in that they repeatedly perform a predefined sequence of actions. Therefore, classes *PRODUCER* and *CONSUMER* inherit from *PROCESS* (see figure 10.3) which provides the basic machinery for a continuous activity; feature *step* in both classes is effected to specify the sequence of actions performed at each iteration.

A GUI version has been implemented by Erol Koç [77]. The source code and the executable can be found on the SCOOP project page.

10.1.3 Binary search trees: efficient parallelisation

The efficiency of software using shared data structures can often be increased by parallelising costly operations on the shared data, so that clients incur smaller delays on access. Due to their shape, lists and tree-like data structures are particularly good candidates for parallelisation.

The binary tree example in OOSC2 ([94], p. 1007) illustrates a simple scheme for concurrent evaluation of the number of nodes in a binary tree. Each tree node has two separate subtrees (possibly void); the computation uses recursive calls on these subtrees. To take full advantage of the potential concurrency, a single blocking query call is replaced by a non-blocking command call (to launch the computation on a subnode) followed by a blocking call to retrieve the result.

This parallelisation pattern may be generalised and applied to the more challenging example — binary search trees (BSTs) — illustrated in figure 10.6. Our implementation exploits to the maximum the potential parallelism of the insertion operation. Clients using binary search trees are not blocked when calling *put*. The non-blocking semantics is achieved through the use of a detachable formal argument in *put*; this avoids locking of the stored element at the time of the call (but the element needs to be locked later on for comparison purposes) as well as the lock passing between the client and the tree (should a client already hold a lock on the stored element), respectively the lock passing between the tree and its subtrees in the recursive calls to *put*. Moreover, a node on which *put* has been called becomes free immediately after executing *store*, without waiting for the termination of the potential recursive call on a subtree. This minimises the contention; if several clients are trying to use the tree at the same time, each of them will only lock the root node for a short moment, and let other clients access the tree as soon as possible. The initiated insertion operations may proceed in parallel. There is no contention if

```
class BINARY_SEARCH_TREE [G -> separate COMPARABLE]
create
    make
feature {NONE} -- Initialization
    make (v: G)
        do
            item := v
        ensure
            item = v
        end

feature -- Basic operations
    item: G
        -- Element stored in current node

    put (v: ?G)
            -- Put v in current tree.
        require
            v_not_void: v /= Void
        do
            if {w: G}v then store (w) end
        end

feature {NONE} -- Implementation
    store (v: G)
            -- Store v.
        do
            if v < item then
                if {l: separate BINARY_SEARCH_TREE [G]}left then
                    subtree_put (l, v)
                else create left.make (v)
                end
            elseif {r: separate BINARY_SEARCH_TREE [G]}right then
                subtree_put (r, v)
            else create right.make (v)
            end
        end

    subtree_put (a_subtree: separate BINARY_SEARCH_TREE [G]; v: G)
            -- Store v in a_subtree.
        do
            a_subtree.put (v)
        end
end
```

Figure 10.6: Parallelised binary search tree

```
-- in class BINARY_SEARCH_TREE
has (v: G)
        -- Does current tree contain v?
    do
        Result := v.is_equal (item)
            or else (v < item and then
                {l: separate BINARY_SEARCH_TREE [G]}left and then
                    subtree_has (l, v))
            or else ({r: separate BINARY_SEARCH_TREE [G]}right and then
                subtree_has (r, v))
    end

subtree_has (a_subtree: separate BINARY_SEARCH_TREE [G]; v: G): BOOLEAN
        -- Does a_subtree contain v?
    do
        Result := a_subtree.has (v)
    end
```

Figure 10.7: Implementation of *has*

they follow different branches; the operations following the same branch are performed in the FIFO order.

The implementation of membership test (query *has* in figure 10.7) makes lesser use of parallelism. The recursive call *a_subtree.has (v)* is blocking; the current node needs to wait until the chosen subtree has terminated the evaluation of that query. In the meantime, the client which called *has* on the root node is blocked waiting for the result; no other clients may access the root node in the meantime. Compared to *put*, the amount of parallelism exhibited by *has* is minuscule. But even this implementation enables some concurrency: subtrees rooted at nodes not involved in the recursive evaluation of *has* are available for use by other clients. For example, if the sought element is smaller than the item stored in the root node, the evaluation of *has* follows the left subtree; although the root node and one branch of the left subtree remain locked, the right subtree and the unused branches of the left subtree are not locked and can be accessed by other clients in the meantime.

The binary search tree example highlights some advanced features of SCOOP.

- The compact implementation is only possible thanks to the full support for genericity. The formal generic parameter G in class *BINARY_SEARCH_TREE* is constrained to $(!, \top, COMPARABLE)$. This is necessary for the application of comparison operators on stored elements, and for storing elements of non-expanded types. (The SCOOP_97 binary tree example from OOSC2 only works for expanded types; the lack of appropriate rules for genericity results in creation of traitors if actual generic parameters of reference types are used.)

- The refined semantics of detachable and attached types (see section 7.1.2) offers the choice between locking and non-locking behaviour of feature application. The operation *has* locks its argument; *put* does not. Despite the use of a detachable formal argument,

put does not accept void actual arguments: the precondition *v* /= **Void** prohibits such arguments.

- The object test mechanism is used for casting detachable entities to attached types. As pointed out in section 6.7, a successful object test with a separate source does not result in locking the source.

Similar parallelised algorithms may be implemented for linked lists, queues, and other data structures. We do not discuss them here; see Erol Koç's semester thesis [77] for a number of parallelised algorithms for two-dimensional arrays implemented in SCOOP.

10.1.4 Santa Claus: barriers and priority scheduling

The classic scenario proposed by John Trono [140] involves a number of synchronisation patterns; it is a good testbed to demonstrate the flexibility (or lack thereof) of a concurrent programming model. We have used this example as part of the final project in the concurrency course focussed on SCOOP (see chapter 12). Let us recall the (slightly embellished) scenario.

> Santa lives in his little house in Jÿëvväskÿølgbrø. He sleeps most of the time, except for rare moments when a group of elves or a group of reindeer wakes him up to perform some activity. Ten elves living in the area produce toys. Once in a while they run out of ideas and need to ask Santa for help. They can only wake up Santa if they form a group of three individuals. There are also nine reindeer living in the stable nearby; Santa uses them to deliver toys. When the reindeer want to work, they need to form a group of nine (that is all the reindeer must join) and wake up Santa. Santa does the delivery round with the reindeer and then goes back to sleep; so do the reindeer. In a situation when a group of elves and a group of reindeer are trying to wake up Santa at the same time, the reindeer get the priority.

Several kinds of synchronisation may be identified here: mutual exclusion (Santa may serve at most one group at a time), barrier synchronisation (a group must be formed before waking up Santa), and priority scheduling (reindeer are serviced before elves). Additionally, there are two potential sources of starvation: on one hand, the reindeer could overtake the elves indefinitely; on the other hand, some elves could be starved by other elves and never get a chance to see Santa.

Several solutions have been proposed in the literature [140, 23]; however, they are convoluted due to the explicit use of low-level synchronisation primitives such as semaphores. The SCOOP solution presented here is simpler. We discuss the implementation of the above synchronisation patterns, and skip the unnecessary details; the full code is available on the SCOOP project page.

The design is highly decentralised, with Santa, elves, and reindeer modelled as separate objects, each of them placed on a different processor; group objects are also separate. Santa and the group objects are passive, the execution is driven by the elves and the reindeer; consequently, classes *ELF* and *REINDEER* inherit from *PROCESS* (via *SANTAS_HELPER*). Figures 10.8, 10.9, and 10.10 show the implementation details. Barrier synchronisation is straightforward: it amounts to calling *join* followed by *leave*, with the appropriate group object as actual argument. The precondition of *join* blocks the client if the group has already been formed; the client

```
deferred class SANTAS_HELPER inherit PROCESS
feature {NONE} -- Implementation
    step -- Basic activity
        do
            do_your_stuff ; join (group); leave (group)
        end

    join (a_group: separate GROUP)
            -- Join a_group.
        require
            a_group. is_join_phase
        do
            a_group. increase_count
            if a_group. is_full then
                a_group. set_is_join_phase (False); wake_up (santa)
            end
        ensure
            a_group.count = old a_group.count + 1
        end

    leave (a_group: separate GROUP)
            -- Leave a_group.
        require
            not a_group. is_join_phase
        do
            a_group.decrease_count
            if a_group.is_empty then a_group. set_is_join_phase (True) end
        ensure
            a_group.count = old a_group.count - 1
        end

    santa: separate SANTA -- Santa
        once
            create Result
        end

    do_your_stuff deferred end -- Do your own stuff.
    wake_up (a_santa: separate SANTA) deferred end -- Wake up a_santa.
    group: separate GROUP deferred end -- Group to join before accessing Santa
end
```

Figure 10.8: Santa's helper

```
class ELF inherit SANTAS_HELPER
feature {NONE} -- Implementation
    group: separate GROUP
            -- Group to join before accessing Santa.
        once
            create Result.make (3)
        end

    do_your_stuff
            -- Produce toys.
        do
            ...
        end

    wake_up (a_santa: separate SANTA)
            -- Wake up a_santa.
        do
            a_santa. ask_for_ideas
        end
end
```

Figure 10.9: Elf

```
class REINDEER inherit SANTAS_HELPER
feature {NONE} -- Implementation
    group: separate GROUP
            -- Group to join before accessing Santa.
        once
            create Result.make (9)
        end

    do_your_stuff
            -- Sleep for a while.
        do
            sleep (...)
        end

    wake_up (a_santa: separate SANTA)
            -- Wake up a_santa.
        do
            a_santa. deliver_toys
        end
end
```

Figure 10.10: Reindeer

must wait until all the members of the current group have left (this is signalled by *is_join_phase*). When the client succeeds, it increases the number of group member and checks whether the group is full; if yes, Santa is accessed through the call to *wake_up*. The mutual exclusion on Santa is achieved simply by passing Santa as actual argument of that call; SCOOP's locking policy enforces exclusive locking of the corresponding processor. Setting *is_join_phase* to **False** lets the group members leave the group (until now, they have been blocked by the precondition of *leave*). The last member leaving the group sets *is_join_phase* to **True**, thus allowing the formation of the next group. The clients blocked on the precondition of *join* start joining, the last one wakes up Santa, and so on.

This implementation satisfies all the requirements expressed in the problem description: elves and reindeer perform their actions in the correct order, elves access Santa only in groups of three individuals, all the reindeer group before accessing Santa, and Santa services at most one group at a time. Additionally, SCOOP's FIFO scheduling policy of satisfiable requests ensures the lack of starvation. An elf cannot overtake another one, i.e. elves trying to form a new group have the priority over those leaving the current group. If a group of elves is trying to access Santa when he is busy with the reindeer, the elves are guaranteed to be served next, even if the reindeer regroup very quickly after being released.

One element of the original problem has not been addressed yet: priority scheduling between a group of elves and a group of reindeer that try to access Santa concurrently. In the presented solution, there is no need for prioritisation because Santa is either busy already servicing one group, or idle and ready to service the next group immediately. In the latter case, a group requesting his service proceeds immediately, so it is never the case that two groups try to access Santa at *exactly* the same time. (One can always decide which group came first.) Therefore, the priority requirement for the reindeer is satisfied vacuously.

To spice up the example, let's assume that Santa may be in an additional state, e.g. performing the *sleep_deeply* operation, where his handler is busy so that groups requesting his services may be blocked for some time. This introduces the possibility of a reindeer group and an elf group vying for the access to Santa at the same time, and requires an explicit handling of priorities. The necessary modifications of the SCOOP solution are minimal: one additional object — a waiting room — is needed; the call to *wake_up* in $\{ELF\}.join$ must be wrapped in a call to *wake_up_with_low_priority* that ensures correct prioritisation; the corresponding call in $\{REINDEER\}.join$ needs no wrapping but the reindeer must signal their presence in the waiting room. Figure 10.11 summarises the modifications.

To wrap up the discussion, let's have a closer look at the convenient use of once functions in this example. There are three such functions: *santa*, *group*, and *waiting_room*. All of them have a separate result type; consequently, their semantics is *once per system*, i.e. they are shared by all instances of the declaring class (see section 9.4). Since *santa* and *waiting_room* are declared in *SANTAS_HELPER*, they are shared by the elves and the reindeer. On the other hand, *group* is implemented in classes *ELF* and *REINDEER*; as a result, all the elves share one group, and all the reindeer share one, but the elves' group is different from the reindeer's.

10.2 Agents and asynchrony

The advantages brought by separate agents (see section 9.3) are exploited here to provide solutions to a number of interesting problems often occurring in practical concurrent applications:

```
-- in class SANTAS_HELPER
    waiting_room: separate WAITING_ROOM
            -- Waiting room for groups.
        once
            create Result
        end

-- in class ELF
    join (a_group: separate GROUP)
            -- Join a_group.
        do
            a_group.increase_count
            if a_group.is_full then
                a_group.set_is_join_phase (False)
                wake_up_with_low_priority (santa, waiting_room)
            end
        end

    wake_up_with_low_priority (a_santa: separate SANTA;
                               a_waiting_room: separate WAITING_ROOM)
            -- Wake up a_santa if there are no reindeer in a_waiting_room.
        require
            not a_waiting_room.reindeer_present
        do
            wake_up (a_santa)
        end

-- in class REINDEER
    join (a_group: separate GROUP)
            -- Join a_group.
        do
            a_group.increase_count
            if a_group.is_full then
                a_group.set_is_join_phase (False)
                signal_presence (waiting_room, True)
                wake_up (santa)
                signal_presence (waiting_room, False)
            end
        end

    signal_presence (a_waiting_room: separate WAITING_ROOM; b: BOOLEAN)
            -- Signal presence or absence of reindeer in a_waiting_room.
        do
            a_waiting_room.set_reindeer_present (b)
        end
```

Figure 10.11: Priority scheduling

parallel evaluation of multiple queries, using a pool of resources for load balancing, rendezvous-style synchronisation and emulation of active objects, and full decoupling of publishers and subscribers in event-driven architectures.

10.2.1 Rendezvous synchronisation and active objects

Concurrency models based on active objects [5, 7, 6, 119] use the rendezvous mechanism to synchronise calls between objects. The asynchronous call mechanism introduced in section 9.3.4 may be used to implement rendezvous-style synchronisation. Simulating active objects is not a goal in itself but rather an interesting exercise to demonstrate the flexibility of our model. Figure 10.12 illustrates a common scenario where objects act both as clients and servers. Object *o1* is a client of some server *o2*; at the same time, it acts as a server to clients *o3* and *o4*. Once in a while, *o1* wants to check whether there are any incoming calls, service them, and then continue its own work. This scenario cannot be implemented in SCOOP_97 because no calls can be made on an object whose handler is busy; therefore, *o1* must terminate its activity to accept incoming calls. How can one be sure that *o1* resumes its activity after servicing some incoming calls? It may be achieved if all incoming calls re-executed *o1*'s routine *live*. But this is difficult to enforce; some clients might not be aware of this pattern. But even if we manage to enforce the pattern, what happens if there are no incoming calls at all? *o1* will never get back to life!

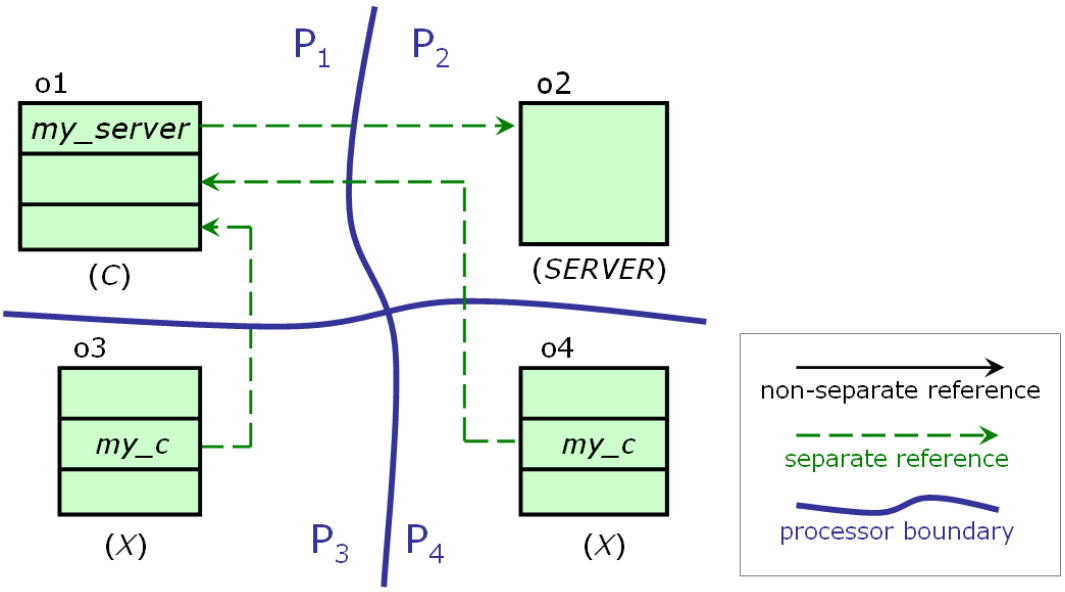


Figure 10.12: Clients-servers scenario

The agent-based mechanism for fully asynchronous calls (see section 9.3.4) provides a solution, illustrated in figure 10.13. An instance of class *C*, e.g. *o1* in our example, behaves like an active object; the routine *live* represents its body. The body is not an infinite loop like in traditional active objects; instead, it is a one-off sequence of calls: a call to *do_work* and an asynchronous call to *live*. Routine *do_work* implements the "client" part of the active object; it includes activities such as calling a server (*o2* in our example), doing some local work, call-

ing the server again, etc. It may contain loops but it must eventually terminate. When *live* terminates, the processor P_1 handling *o1* becomes idle and is unlocked; we may assume that the client which locked it beforehand has released its lock. One of the clients (if any) waiting for *o1* may proceed now: lock P_1 and perform its calls on *o1*. Assuming that *o3* and *o4* issued a request while *o1* was busy, both are serviced one after the other; P_1 is released after servicing each request. This is the "server" part of *o1*'s activity. Eventually, *o1*'s own call to **Current**.*live* issued in the previous iteration of *live* is serviced, "waking up" *o1* to resume its activity. The server phase restarts: *o1* can do its own work again, call other servers, etc. Note that only incoming calls whose wait-conditions are satisfied are serviced; therefore, *o1* may "select" incoming calls. This emulates the

select when => ... **accept** ...

construct of Ada.

```
-- in class C

my_server: separate SERVER

live is
        -- ''Body'' of active object.
    do
        do_work
        asynch (agent live)
    end

do_work is
        -- Active object's own work.
        -- Calling servers, doing local work, etc.
    local
        in_data, out_data: DATA
    do
        in_data := fetched_from_server (my_server)
        out_data := f (in_data, ...)  -- Local calculations
        ...
        update (my_server, out_data)
    end
```

Figure 10.13: Active objects in SCOOP

The above pattern has been turned into a reusable class *ACTIVE_OBJECT*, and included in the *CONCURRENCY* library (see section 11.4 and appendix A). Descendants of *ACTIVE_OBJECT* simply need to effect the deferred feature *do_work* representing the active object's body. Note the similarity between this class and the class *PROCESS* discussed in a previous section. The main difference is that the latter does not provide any facilities for its instances to act as servers; process objects are "pure clients".

10.2.2 Waiting faster

It is possible to optimise the evaluation of query calls on several separate objects by spawning the computations in parallel and combining the partial results as they arrive, in the spirit of Tony Hoare's principle of "waiting faster" [66]. The following code excerpt illustrates a naive attempt at implementing parallel wait in SCOOP. The scenario involves two boolean queries whose results should be evaluated concurrently and combined using the operator **else**.

```
if parallel_or (my_x1, my_x2) then ... end

parallel_or (x1, x2: separate X): BOOLEAN
    do
        Result := x1.q or else x2.q
    end
```

A client calling *parallel_or* wishes to minimise waiting; however, this implementation is sub-optimal.

- The client is blocked until both actual arguments *my_x1* and *my_x2* are free. Even if *my_x1* is busy but *my_x2* is free and *my_x2.q* would evaluate to **True** so that the result of *my_x1.q* could be ignored, the client still has no chance to proceed.
- The evaluation of the boolean expression in *parallel_or* is sequential, due to wait by necessity; *x1.q* must be evaluated before *x2.q*. This is particularly painful if the former call takes more time to terminate than the latter; reversing the evaluation order would be advantageous.
- The **or else** operator only palliates the problem: the evaluation stops if *x1.q* returns **True**; the second call is not executed.

To avoid the first problem, the example may be rewritten as

```
if q (my_x1) or else q (my_x2) then ... end

q (x: separate X): BOOLEAN
    do
        Result := x.q
    end
```

but the other issues remain. The problem becomes even more acute if we want to generalise this scenario to n queries (not necessarily of the same form) applied to a number of targets, and combined by a user-defined combinator (more complex than the basic operators **or**, **and**, **not**, etc.). The wait by necessity principle and the locking semantics of argument passing seem to stand in the way of efficient implementations. This certainly was the case in SCOOP_97; SCOOP, however, offers a simple solution based on agents. Figure 10.14 illustrates the technique (objects depicted in yellow are created implicitly by the mechanism; they are invisible to the client object). Instead of being applied directly by the client, each query is evaluated by an independent evaluator object placed on a freshly created processor. The evaluations proceed in parallel; the incoming partial results are combined by a centralised answer collector (also placed

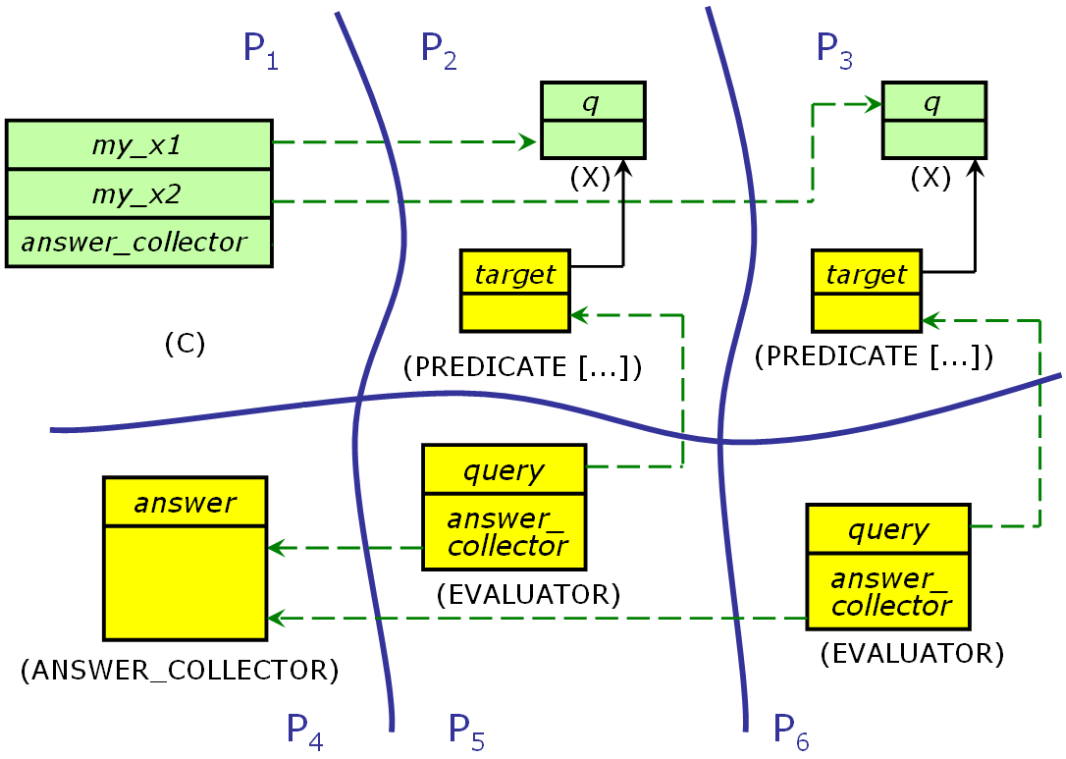


Figure 10.14: Parallel evaluation of boolean queries

on its own processor). If the first incoming result is **False**, the answer collector waits for the second one. On the other hand, if the first partial result is **True**, the client receives the answer immediately, without waiting for the second one. Depending on the availability of *my_x1* and *my_x2* and the evaluation speed of each query, partial results may be reported in any order, e.g. *my_x2.q* may return before *my_x1.q*. Therefore, the solution is fully symmetric and does not suffer from the limitations of the naive solution discussed above.

The implementation is sketched in figure 10.15. All the necessary machinery is implemented in the *CONCURRENCY* library. The client uses the predefined feature *parallel_or* , passing as argument a list of agents representing the queries *my_x1.q* and *my_x2.q* to be evaluated; *parallel_or* itself relies on the function *evaluated_in_parallel* which supports parallel evaluation of any number of queries and combination of their results. Feature *evaluated_in_parallel* creates a separate instance of *ANSWER_COLLECTOR* which takes care of launching the independent evaluation of queries by dedicated instances of *EVALUATOR*. Each evaluator tries to evaluate its query (which involves locking the target's processor, calling the agent, and retrieving its result) and send the result to the answer collector. The answer collector is idle, so the evaluators can lock it and execute its feature *update_result* which combines the current answer with the result provided by the calling evaluator. In the meantime, the client is blocked on the call to *answer* (*answer_collector*); it will proceed as soon as the answer is ready, i.e. either all the partial results have been combined, or the answer has a value that lets the client ignore the remaining results (in our case, the client proceeds as soon as one query evaluates to **True**). Note that programmers are shielded from these implementation details; all the above features come from the library class *CONCURRENCY* and can be readily used in different applications. Full details of the implementation — including classes *CONCURRENCY*, *EVALUATOR*, and *ANSWER_COLLECTOR* — can be found in appendix A; figure 10.15 only

```
-- in class C that inherits from CONCURRENCY
l: LINKED_LIST [?separate PREDICATE [ANY, TUPLE]]
...
create l.make
l.put (agent my_x1.q)
l.put (agent my_x2.q)
if parallel_or (l) then ... end

-- in class CONCURRENCY
parallel_or (l: LIST [?separate PREDICATE [ANY, TUPLE]]): BOOLEAN
    do
        if {res: BOOLEAN} evaluated_in_parallel (l, False, True,
                                agent or_else (b1, b2: BOOLEAN): BOOLEAN
                                    do Result := b1 or else b2 end (?, ?))
        then Result := res end
    end

evaluated_in_parallel (a_queries: LIST [?separate FUNCTION
                                [ANY, TUPLE, ?separate ANY]];
                       an_initial_answer, a_ready_answer: ?separate ANY;
                       an_operator: FUNCTION
                                [ANY, TUPLE, ?separate ANY]): ?separate ANY
        -- Parallel evaluation of queries combined by an_operator
    require
        a_queries.count > 0
    local
        answer_collector: separate ANSWER_COLLECTOR
    do
        create answer_collector.make (a_queries, an_initial_answer,
                                      a_ready_answer, an_operator)
        Result := answer (answer_collector)
    end

answer (an_answer_collector: separate ANSWER_COLLECTOR): ?separate ANY
        -- Answer from an_answer_collector
    require
        an_answer_collector.is_ready
    do
        Result := an_answer_collector.answer
    end
```

Figure 10.15: Parallel or

shows the most important features. The formal arguments of *evaluated_in_parallel* need some explanation:

- *a_queries* is a list of queries to be evaluated. In our scenario, the list contains two agents:

```
-- in class C
my_fancy_operator (l: LIST [?separate FUNCTION
                                [ANY, TUPLE, INTEGER]]): INTEGER
        -- Some fancy parallel operator.
    do
        Result := evaluated_in_parallel (l, ..., ...,
                    agent fancy (x, y: INTEGER): INTEGER
                        do
                            Result := ...
                        end (?, ?))
    end
```

Figure 10.16: User-defined parallel operator

agent *my_x1.q* and **agent** *my_x2.q*.

- *an_initial_answer* is an initial value for the result. The *parallel_or* operation uses **False** as initial value because it is the neutral element for **or else** (that is, **False or else** *b* is the same as *b*).
- *a_ready_answer* indicates an early termination of the evaluation process; if the current answer is equal to that value, it may be given back to the answer collector immediately, without waiting for the remaining partial results. In our case, this value is **True** (because **True or else** *b* is the same as **True**).
- *an_operator* is a user-defined binary operator for combining the current answer with each partial result. In our example, it is the inline agent **agent** *or_else* (?, ?) which implements the **or else** operation. (Inline agents have not been discussed in this dissertation; however, the rules defined in section 9.3 support their safe handling. See the Eiffel standard [53] for details concerning inline agents.)

A number of useful parallel operators — *parallel_or*, *parallel_and*, *parallel_sum* — have been implemented in the class *CONCURRENCY* (see figure A.2 in appendix A). Enriching the library of parallel operators is straightforward; should a user-defined operator be needed, it can be constructed using *evaluated_in_parallel*, as shown in figure 10.16.

10.2.3 Resource pooling

Shared resources are often organised in pools to optimise the efficiency through load balancing. Clients requesting a service want one of the currently available resources but it does not matter which resource is chosen. For example, an incoming phone call must be despatched to the first available operator; the caller does not specify the exact phone line. We would like to write it as

```
pool: LIST [separate PHONE_LINE]
free_line : separate PHONE_LINE
...
free_line := pick_first_available (pool)
connect_to ( free_line )
```

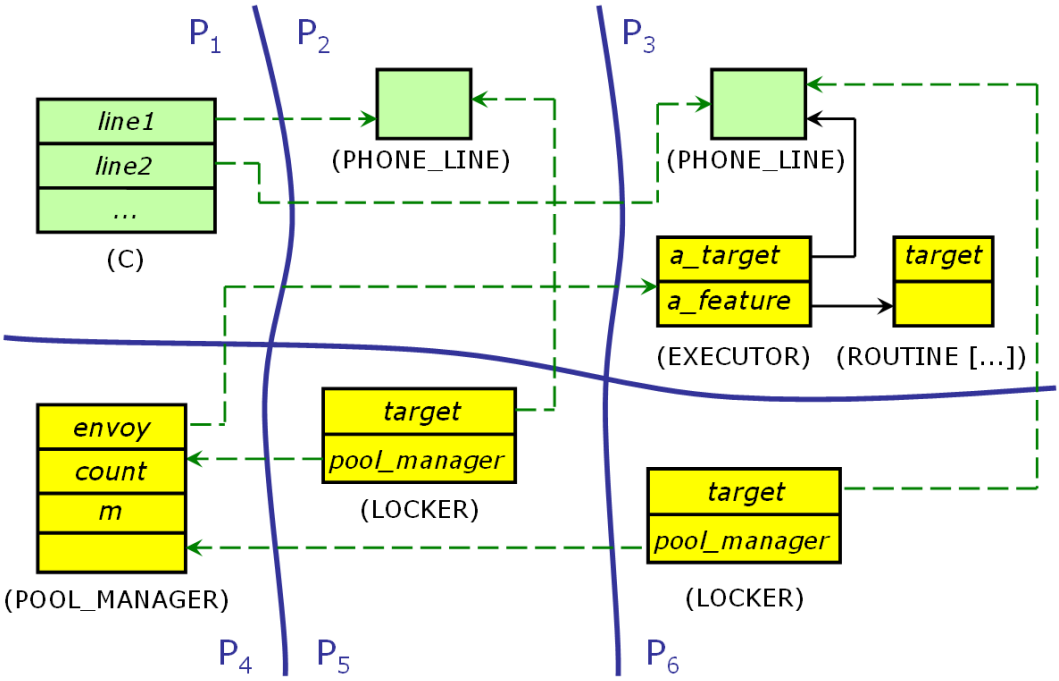


Figure 10.17: Using resource pool

Unfortunately, this will not work. The basic SCOOP model offers no support for a "lock one out of n resources" operation represented by *pick_first_available* ; there is no "try to lock" operation which could be used to implement that facility. But even if it existed, the phone line picked in the first step might become busy before the call to *connect_to*, because the lock is released in the meantime. For the same reason, more than one incoming call might select the same phone line.

Here too, the new mechanisms introduced in SCOOP — agents and lock passing — come to the rescue. The former enables parallel attempts at locking each resource; the latter permits passing a lock held by one processor to another processor. Figure 10.17 depicts the solution (only two instances of *PHONE_LINE* are shown but the solution works for any number of resources). An independent instance of *LOCKER* is created for each phone line; its task is to lock the assigned phone line object and pass the lock to the pool manager which applies the requested feature to the phone line object. Although several lockers may succeed in grabbing a phone line, only the first one reporting to the pool manager is taken into account; the others are ignored. In our scenario, *line2* was locked first, so the requested feature is applied to it. All the objects depicted in yellow are "transparent" to the client; they have been created by the underlying implementation provided as part of the *CONCURRENCY* library. The client code and the essential features from the library are shown in figure 10.18; the library classes can be found in appendix A. The feature *call_m_out_of_n* implements a general mechanism to apply a feature (represented as an open-target agent) to m elements from a resource pool. In our example, feature **agent** {*PHONE_LINE*}.*connect*(**Current**) is applied to one out of n phone lines from *pool*. Recall that open-target agents cannot be used with separate targets; therefore, the pool manager places an "envoy" object (an instance of *EXECUTOR*) chosen phone line's processor. (A qualified processor tag is used to indicate that the envoy be handled by that processor; see figure A.8.) The envoy first obtains a non-separate copy of the agent (using the *import* operation discussed in section 6.8) and then applies the agent to the target (which is

```
-- in class C that inherits from CONCURRENCY
pool: LINKED_LIST [?separate PHONE_LINE]
 ...
create pool.make
pool.put (line1)
pool.put (line2)
 ...

call_m_of_n (agent {PHONE_LINE}.connect (Current), pool, 1)
    -- Call connect on 1 element of pool.

-- in class CONCURRENCY
call_m_out_of_n ( a_feature : ROUTINE [ANY, TUPLE];
                  a_pool: LIST [?separate ANY]];
                  m: INTEGER)
        -- Apply a_feature to m elements of a_pool.
    require
        m > 0 and then a_pool.count >= m
    local
        pool_manager: separate POOL_MANAGER
        locker : separate LOCKER
    do
        create pool_manager.make (a_feature , m)
        from a_pool. start until a_pool. after loop
            create locker. try_to_lock (a_pool.item, pool_manager)
            a_pool.forth
        end
    end
```

Figure 10.18: Locking and calling 1 out of n resources

non-separate from the envoy).

The implementation of resource pooling facility highlights the usefulness of several mechanisms discussed in this dissertation: separate agents, generic data structures, lock passing, and safe import of separate objects. It also shows that certain limitations of the agent mechanism can be overcome by a judicious use of other facilities.

The mechanism presented here can be used for many different purposes: non-deterministic choice of resources, load balancing, choosing the resource with the shortest response time, etc. It may be combined with the previously discussed mechanisms such as fully asynchronous feature calls and "waiting faster"; it may also be used in event-driven applications based on the facilities described in the next section.

10.2.4 Event-driven programming

Event-driven programming has gained considerable popularity over the past few years, in particular in GUI applications where it facilitates the separation of concerns: an application layer (business logic) provides the operations to execute, whereas a GUI layer triggers their execution in response to users' actions; the two layers communicate by publishing and receiving event notifications. Objects that publish events are called *publishers*; objects that receive notifications are referred to as *subscribed objects*.

Arslan et al. [12] propose a compact Eiffel library for event-driven programming. The library consists of a single class *EVENT_TYPE* which provides basic facilities for event publication, subscribing to an event type, and unsubscribing from it. Any object may become a publisher simply by calling the corresponding feature *publish* on a given instance of *EVENT_TYPE*; multiple objects may act as publishers of the same event type. Subscribed objects are agents (of type *ROUTINE* [*ANY*, *TUPLE*]). The same agent may be subscribed to many event types; conversely, multiple agents may be subscribed to the same event type. Every instance of *EVENT_TYPE* keeps a list of subscribed agents; on event publication, the subscribed agents are called in sequence. Figure 10.19 shows a typical scenario with one event type, two publishers, and three subscribed objects. (The subscribed objects are agents; their actual targets, depicted in yellow, are "transparent" to the event mechanism.) Note that all the objects are non-separate from each other.

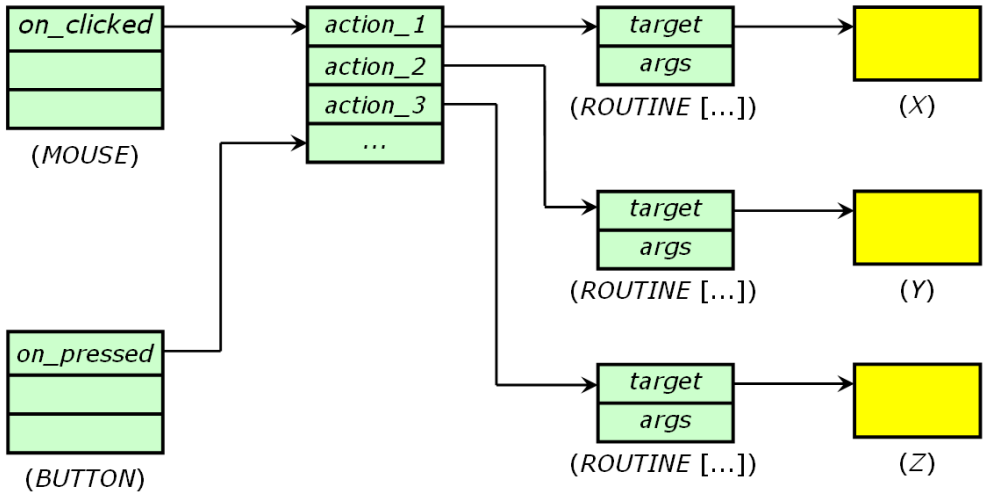


Figure 10.19: Event-driven programming with original Event library

The Event library provides *space decoupling*, i.e. publishers and subscribed objects need not know each other's identity; they do not even know how many publishers and subscribed objects participate in the interaction. The authors suggest the use of SCOOP to provide *flow decoupling*, i.e. ensure that event publication be non-blocking for the publishers, and that subscribed objects need not actively poll for events but are free to do their own work in the meantime. Nevertheless, no SCOOP-compliant implementation has been proposed. Similarly, it is suggested that SCOOP should enable full *time decoupling*, whereby the notification of different subscribed objects may happen at different times. The library clearly fails to address the last issue: if two

subscribed objects **agent** $x.f$ and **agent** $y.g$ occur in the list one after the other, the latter will not be applied before the former has terminated.

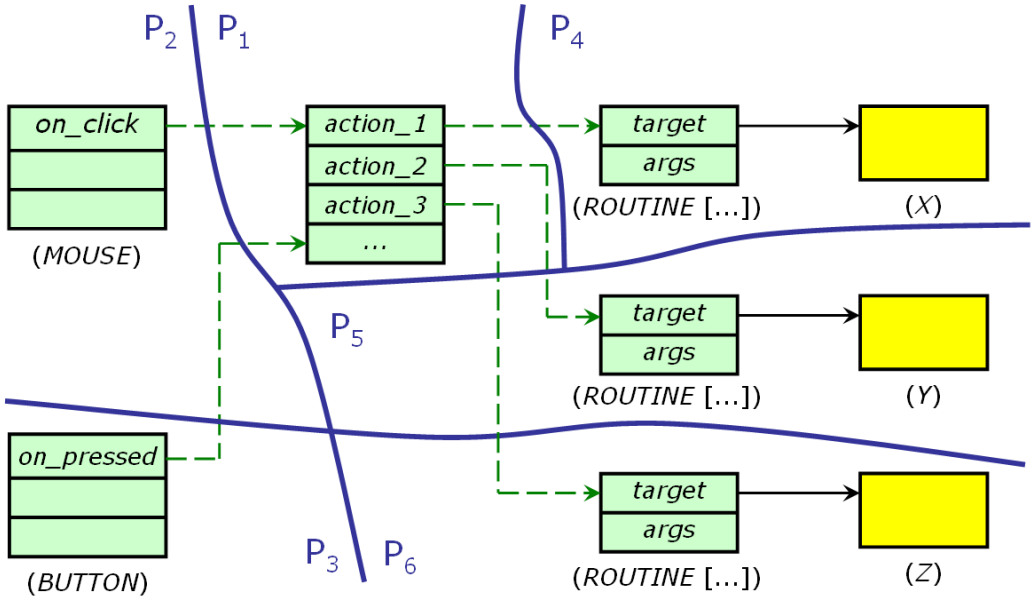


Figure 10.20: Event-driven programming with SCOOP-enabled Event library

We have reimplemented the Event library to make it compatible with SCOOP and to provide support for flow decoupling and time decoupling. Figures A.10–A.11 show the modified class *EVENT_TYPE*. The major changes with respect to the original library are:

- The constraint on the formal generic parameter *EVENT_DATA* is relaxed to $(?, \top, TUPLE)$; the original class imposed a stronger constraint: $(!, \bullet, DATA)$.
- The underlying data structure keeping a list of subscribed objects is now typed as

  LINKED_LIST [?**separate** *ROUTINE* [*ANY*, *EVENT_DATA*]]

  thus allowing subscription of separate agents. Accordingly, features *subscribe* and *unsubscribe* take an argument of that type; its detachability prevents locking when subscribing or unsubscribing an agent.
- Rather than calling all subscribed agents in sequence (thus blocking at every step until the call has terminated), the body of *publish* performs a series of fully asynchronous calls, one for each subscribed object.

```
from start until after loop
    if {action: separate ROUTINE [ANY, EVENT_DATA]}item then
        asynch (agent action.call (arguments)) end
    forth
end
```

  This is achieved through the *asynch* facility provided by the library class *CONCURRENCY* (see section 9.3.4). An object test is necessary to downcast the currently processed subscribed object to an attached type, in order to build an agent expression on it. Note the interesting double wrapping of subscribed actions: *action* is itself

an agent; a call to it is wrapped in another agent and passed to *asynch*. This is needed to include the actual arguments *arguments*; an alternative (more complex but potentially more efficient) solution would be to extend the *CONCURRENCY* with a version of *asynch* taking two arguments and to provide a corresponding feature in class *EXECUTOR* (see appendix A).

Figure 10.20 depicts the previous scenario augmented with SCOOP features. Each participating object is now handled by a different processor; the asynchronous communication between the publishers and the event type on one hand, and between the event type and the subscribed objects on the other hand allows full flow decoupling. Full time decoupling is achieved through *asynch* calls on each subscribed objects. As a result, none of the publishers need to wait until the event type starts the notification process. Similarly, the notification of each subscribed object proceeds independently; if one of these objects is not available, this has no influence on the others or on the event type.

To achieve its goals, the proposed extension of *EVENT_TYPE* combines several techniques proposed in this dissertation: enriched type system (see chapter 6), new semantics of detachable types (see chapter 7), support for constrained genericity and agents. It also relies on the *CONCURRENCY* library presented in the next chapter.

10.3 Control systems

The growing popularity of O-O techniques in robotics provided part of the impetus for the work on SCOOP. Right from the beginning, we planned to exploit the modelling power of object technology and DbC and apply our model to programming control systems. We present here two such systems written in SCOOP: a highly parallel elevator control system, and an arm robot which sorts items from a conveyor belt according to their colour and places them in the corresponding bins.

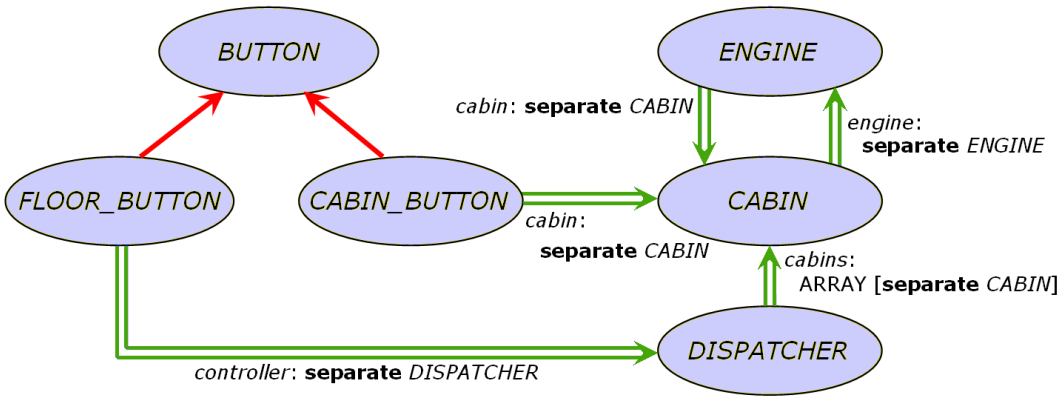
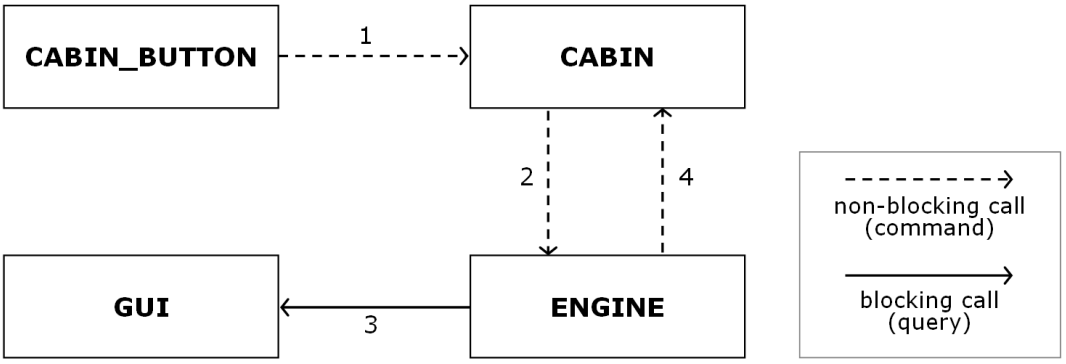


Figure 10.21: Elevator: class diagram

10.3.1 Elevator

The elevator example ([94], p. 1014) describes software for a multi-elevator system serving many floors. The proposed algorithm aims at achieving the maximum decentralisation of coop-

erating objects. In the author's own words, it is "somewhat fanatically object-oriented" in that every significant component — even each individual button in an elevator cabin — is modelled as a separate object with its own handler. Components communicate principally through command calls to take advantage of the asynchronous character of such communication. The major benefit of this architecture is the event-driven nature of object interaction; no loops are needed to poll the status of objects.



SCENARIO: pressing cabin button to move cabin
1 Cabin button calls *cabin.accept (floor)*
2 Cabin calls *engine.move (floor)*
3 Engine calls *gui.animated (current_floor, floor)*
4 Engine calls *cabin.record_stop (floor)*

Figure 10.22: Elevator: interaction between objects

Figure 10.21 shows the relationships between application classes (excluding the GUI components). Figure 10.22 depicts the interaction between a cabin button, the corresponding cabin, its engine, and the GUI window, in a situation where the cabin button has been pressed. The four depicted objects are placed on different processors. As a result, command calls (dashed lines) between the objects are non-blocking, e.g. right after issuing a call to the engine (step 2), the cabin becomes idle and is ready to receive subsequent requests from cabin buttons or a notification from the engine (step 4). On the other hand, the query call (solid line) from the engine to the GUI component is blocking, i.e. the engine waits until the execution of *gui*. *animated* (*current_floor* , *floor*) has terminated, before proceeding to step 4. Although we omit the implementation details here, figure 10.22 should give a good indication of the general style of the solution. For more details, see the SCOOP project page or one of the student reports mentioned below.

The OOSC2 design has been used, with little modifications, in a number of implementations:

- *Elevator.NET* application by Arslan and Nienaltowski.
  It combines a business logic part written in Eiffel and $YO\_SCOOPLI$ (see section 4.3) with a GUI written in C♯/WinForms, and targets the Microsoft .NET 1.1 platform.

  No supporting tools were available for SCOOP at that time; therefore, calls to SCOOPLI features are woven directly into the source code. Currently, SCOOP applications target-

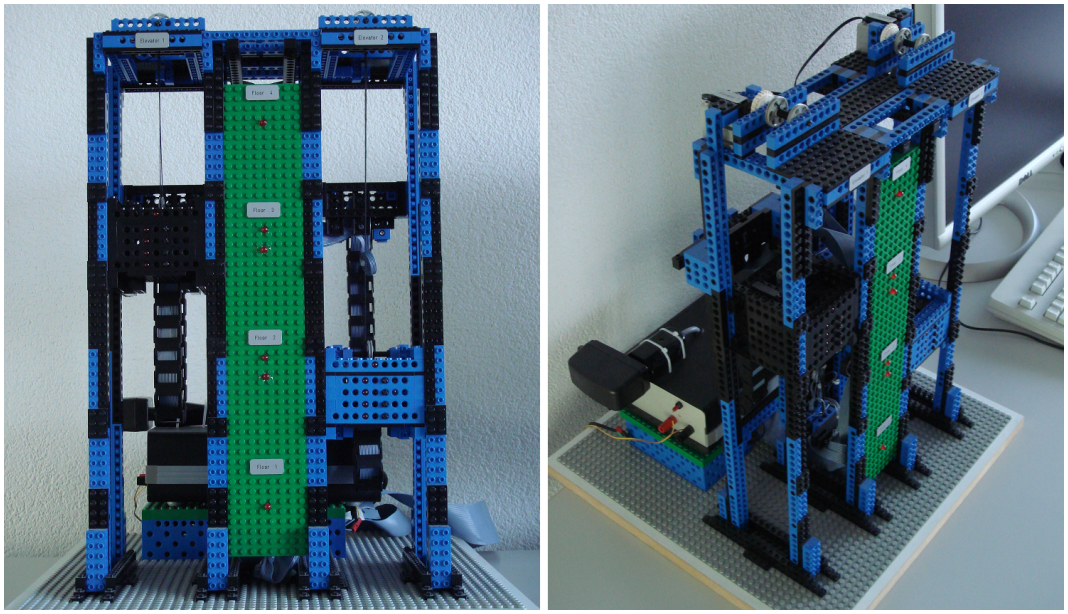


Figure 10.23: Physical model of a double-shaft elevator

ing .NET can be preprocessed using *scoop2scoopli*; no manual code modifications are necessary (see section 11.3 for details).

- *Elevator* application by Erwin Betschart [24], using SCOOPLI and EiffelVision graphical library.
  Here too, SCOOPLI features are used directly in the source code.

- *Physical elevator* built by Matthias Humbert as part of his Diploma thesis [67].
  The physical model consists of two elevator shafts, two cabins with individual engines, and four floors (see figure 10.23). Each cabin has four buttons inside; floors have "up" and "down" buttons, except the bottom and the top floor which only have one button. Due to the presence of directed floor buttons, the despatching algorithm is more sophisticated than the one described in OOSC2; however, no further optimisations are introduced, i.e. pressing a floor button may result in sending a request to the cabin which is currently moving even if the other cabin is idle. (One could think of reimplementing the algorithm using the resource pooling facilities described in section 10.2.3, to optimise the use of cabins and minimise waiting time.)

  The control software is based on SCOOPLI; it communicates with the physical elements through a custom-made library of C routines, wrapped in Eiffel's **external** features. See the project report [67] for details.

The software examples (source code and executables) can be found on the SCOOP project page; a short demo of the physical elevator is also available there.

10.3.2 Arm robot

SCOOPbot is a direct offspring of the concurrency course described in chapter 12. The arm robot has been developed by Ganesh Ramanathan — one of our students and, at the same time,

employee of a company specialised in automation of production systems — who decided to explore SCOOP's potential in robotics. We describe here SCOOPbot's capabilities and give an overview of its control software; more details can be found in the project report [126].

Functionality

The robot, depicted in figure 10.24, searches for items on a conveyor belt, picks them up (one at a time), and puts them in bins, according to their colour. SCOOPbot is composed of a flexible arm, a turntable, a claw, three motors for activating these components, and a set of sensors: an angle sensor for detecting the position of the turntable, two sensors for detecting the position of the arm and the claw respectively, and a light sensor for item recognition. All these elements come from the Lego MINDSTORMS^{TM} framework.

A startup sequence consists of three operations performed in parallel:

- *initialise turntable*
  Find start and end positions; define the positions of different bins; place the turntable in the start position.
- *initialise arm*
  Fully lower then lift the arm; leave it in the top position.
- *initialise claw*
  Close then open the claw; leave it fully opened.

After the initialisation the robot starts seeking objects on the production path by turning the turntable between the start and the end position. If no objects are found before reaching the end position, seeking continues in the opposite direction; then, if no objects are found before reaching the start position, the sequence is restarted. When an object is found, the arm is lowered, the object is grabbed by the claw and picked up. The object is transported to the bin corresponding to its colour. The presence of an object and its colour are detected by the light sensor. The seeking sequence is repeated until a stop is requested; in that case, the object currently held in the claw is put back on the production path, and all the actuators move to their start position. If an emergency stop is requested, e.g. due to the blocked turntable, all the actuators are immobilised immediately after placing the currently held object on the path.

Control software

The mapping between the physical components of the robot and the software objects is straightforward; essentially, each component is represented by an object placed on an independent processor. Figure 10.25 illustrates this architecture; it shows the controller and the actuators (sensors are omitted). The objects modelling the physical components provide the following basic services:

- Turntable: *initialise* , *seek_object* , *move_to_bin*, *move_to_start* .
- Arm: *raise* , *lower*.
- Claw: *open*, *close* .

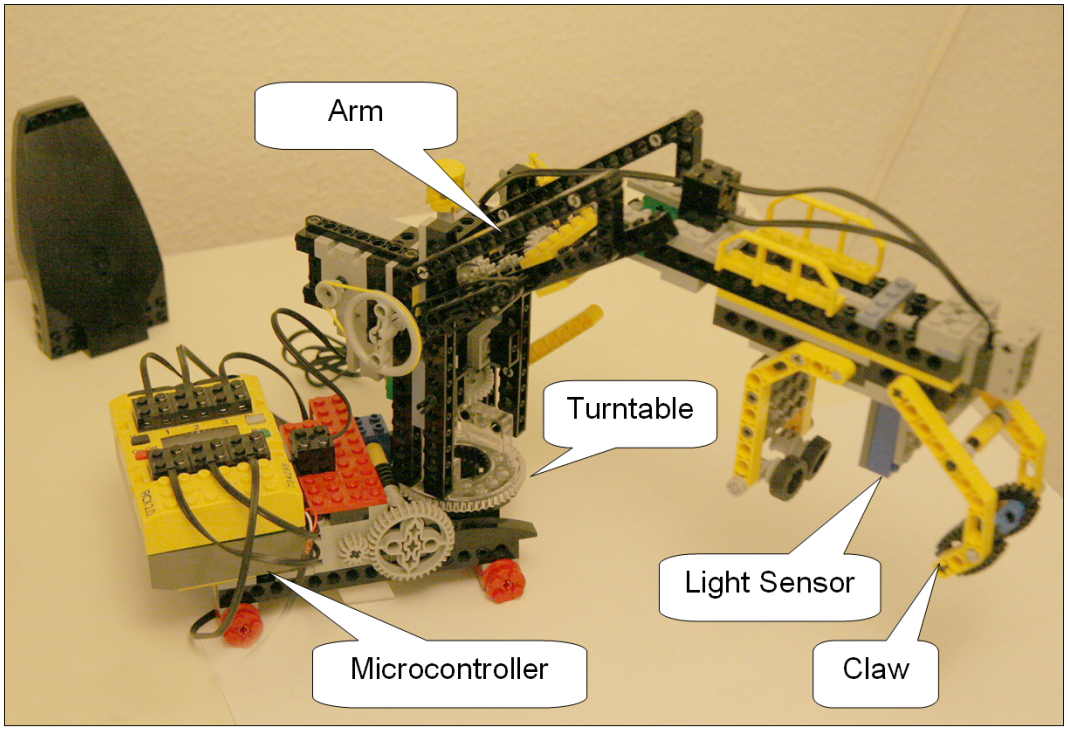


Figure 10.24: SCOOPbot arm robot

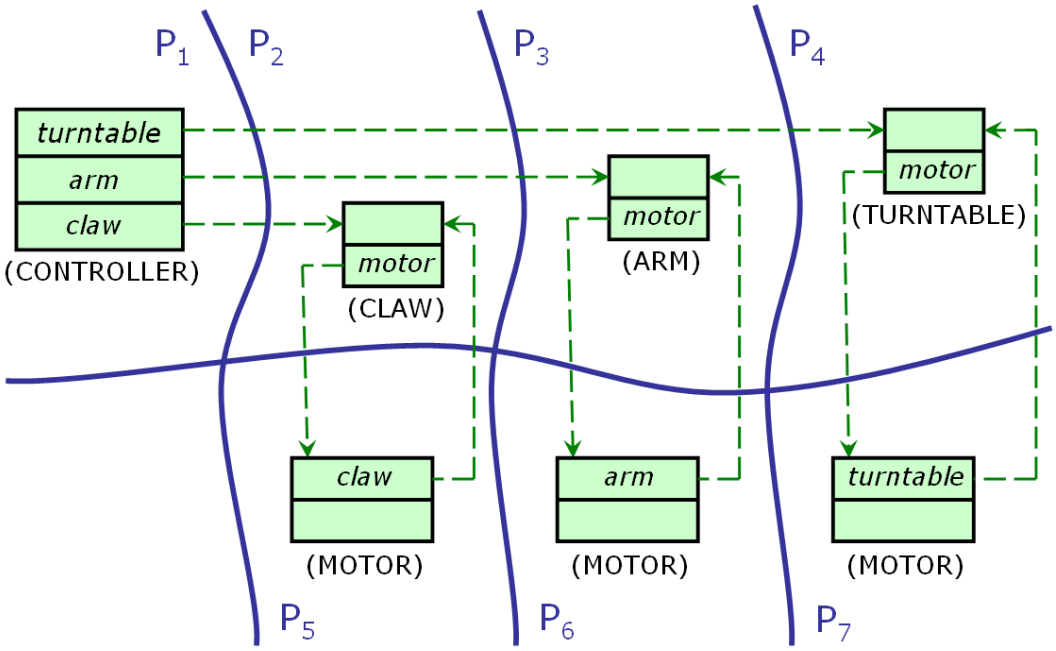


Figure 10.25: Software representation of SCOOPbot's components

- Motors: *run*, *stop*.
- Sensors: *get_value* , *clear*.
- Controller: *run*, *stop*, *emergency_stop*.

Similarly to the elevator example discussed earlier, the design of SCOOPbot is massively parallel. The communication between objects is predominantly asynchronous; however, due to additional requirements, such as prioritisation of requests and acknowledgment of service completion, command and query calls supported by the basic SCOOP model are not sufficient; a richer mechanism is needed. In the current control software for SCOOPbot, explicit messaging is used; the necessary facilities are provided by classes *MESSAGE* and *SERVICE_PROVIDER*. All the objects representing physical components are instances of *SERVICE_PROVIDER*. They keep a queue of incoming requests and process them according to their priorities. Requesting a service requires the creation of a message (an instance of *MESSAGE*) and putting it into the target's request queue. If an acknowledgment of service termination is required, it is also sent as an explicit message.

We are currently migrating the communication infrastructure to agent-based implicit messaging. Thanks to an *asynch*-like mechanism enriched with callbacks and priorities, explicit messages and request queues are not necessary; objects can now communicate directly using feature calls. For example, the controller may ask the turntable to move to a bin in the following manner

```
request_with_ack (agent turntable.move_to_bin, -- service
        Urgency_normal, -- priority
        agent notify_turntable_reached_bin , -- ack callback
        agent turntable.stop) -- emergency operation
```

Interfacing with physical components

All physical sensors and actuators communicate with the control software via the MINDSTORMS RCX controller. A COM wrapper provides an interface between the controller and the SCOOP application. The application runs on the Microsoft .NET platform; this is possible thanks to the support for .NET offered by $scoop2scoopli$ (see section 11.3). Version 5.6 of the ISE Eiffel compiler, which was used for compiling the control software, did not have direct support for COM; hence the use of .NET. With the arrival of EiffelStudio 5.7 and the EiffelCOM library, the control software can be run directly on native Windows.

Discussion

The SCOOPbot project explores O-O modelling of physical components, with a particular focus on the distribution of interacting objects across multiple processors, and the use of contracts for synchronisation. The first version of the control software uses explicit messaging to achieve asynchronous communication with callbacks; the next version will rely on implicit messaging based on an extension of the *asynch* mechanism that provides callback facilities and priorities. SCOOPwash, the next robotics project led by the same author, aims at exploiting the advanced

O-O techniques offered by SCOOP to implement more complex adaptive control algorithms used in automated car wash systems. Another important goal is to achieve a high amount of software reuse between the SCOOPbot and SCOOPwash, and discover further patterns which, like the implicit messaging facility in SCOOPbot, may be turned into reusable components.

10.4 Code reuse

Object technology fosters reuse; in many COOLs, however, it is hindered by the apparent clash between synchronisation and O-O mechanisms, in particular inheritance.

Multithreading models of popular languages such as Java and C♯ offer limited support for sequential code reuse. A direct reuse of library classes that have not been designed for concurrency often leads to data races and atomicity violations. The supplier-side (server-side) synchronisation policy used in these models requires the classes used in a concurrent application to be written "with concurrency in mind". Any feature that may be accessed simultaneously by several threads needs to implement an appropriate synchronisation mechanism, usually in the form of a

while (*someCondition*) *wait*();

loop. Since it is difficult to cater for all the future contexts in which a class may be used, programmers often implement libraries in a defensive style, providing additional synchronisation "just in case". This results in heavy, entangled code which is difficult to extend and reuse.

SCOOP simplifies the reuse of existing libraries, be they written for sequential or concurrent systems in the first place. Most importantly, all classes available in standard libraries can be used in concurrent programs without any modifications. The mutual exclusion guarantees offered by the model, coupled with the new semantics of contracts, ensure a correct synchronisation of clients' calls, thus letting programmers focus on the important part of their work rather than bothering about potential atomicity violations. This leads to clearer code which can be easily reused and extended through inheritance. A sequential class, e.g. *BOUNDED_QUEUE*[*G*], can be taken and used directly in a concurrent application

my_buffer: **separate** *BOUNDED_QUEUE* [*INTEGER*]

Declaring *my_buffer* as separate suffices to turn the bounded queue into a bounded buffer; no modifications of the base class *BOUNDED_QUEUE* are necessary.

The sequential-to-concurrent code reuse discussed above was already supported by SCOOP_97 (except for the use of genericity); however, a number of limitations existed (see section 5.12.3). In SCOOP, these limitations do not apply, thanks to the unified contract semantics and the enriched type system. For example, the initially problematic reuse of class *STRING* in a concurrent context (see figure 5.16) is not hindered any more: precise type annotations with qualified processor tags permit safe handling of this and similar scenarios. Figure 10.26 illustrates the approach. Since *s* and *s2* are statically known to be non-separate from each other, the call *s*.*append* (*s2*) is valid. The implicit type rule 6.2.3 permits correct type-checking of the second call *s*.*append* (*s*). Class *STRING* is not an exception; this may be just a toy example but it illustrates well the general technique of accommodating library classes whose features take non-separate formal arguments but are used on separate targets.

Unlike many other approaches, including the multithreaded models of Java and C♯, SCOOP

```
r (s: separate STRING)
    require
        not s.is_empty
    local
        s2: separate <s.handler> STRING
    do
        s2 := s.twin
        s.append (s2)
        s.append (s)
    end
```

Figure 10.26: Sequential-to-concurrent reuse

also supports a straightforward concurrent-to-sequential reuse, i.e. the code written for a concurrent application can be safely used in a sequential context. This may seem obvious at first but is not a trivial problem, in particular in the presence of condition synchronisation. For example, a Java class implementing a shared buffer may not work properly in a sequential context because the (unique) thread performing the operation *put* is blocked if the buffer is full; it waits for other threads to wake it up but no other thread ever accesses the buffer.

```
// Buffer <G>
public synchronized void put (G value) {
        while (isFull()) wait();
        queue.addLast(value);
        count++;
        notifyAll();
    }
```

To avoid this problem, we need to use a different version of class *Buffer* that implements non-shared buffers, or at least a different version of feature *put*. In SCOOP, the same class *BUFFER* and feature *store* may be safely used in both context, although they are primarily destined for concurrent applications. The contract of *put* captures the conditions of correct use:

```
-- in class BUFFER [G]
put (v: G)
    require
        not is_full
    ensure
        count = old count + 1
        last_item = v
```

Clients using the buffer must respect this contract. They may achieve it by using an appropriate wrapping routine, e.g.

```
-- in class CLIENT
store (buffer: separate BUFFER [INTEGER]; v: INTEGER)
    require
        not buffer.is_full
    do
```

```
        buffer.put (v)
    end
```

The precondition of *store* ensures the necessary condition synchronisation; the client is blocked until the buffer becomes non-full. Even though *my_buffer* is not separate from the client — which is precisely the same situation as in the Java example with a single thread — the call to *store* (*my_buffer*, 3) will not deadlock. The generalised semantics of preconditions reduces to the correctness semantics here, so a call on a full buffer gives raise to a precondition violation, thus giving the client a chance to handle the situation rather than blocking forever. Concurrent code, when used in a sequential context, behaves just like sequential code; this is the essence of concurrent-to-sequential reuse in SCOOP.

10.5 Inheritance anomalies

Problems arising from the combination of inheritance and concurrency have been observed since the early eighties. In most O-O concurrency models they are caused by a high interdependence between the attributes of a class and the coordination constraints of different routines. Concurrency coordination and functional code are usually interwoven; as a result, features cannot be redefined in subclasses without affecting other features. Very often, the affected features must be redefined in the descendant and the ancestor, which degrades the maintainability and prevents the reuse of code. Even if the coordination code is isolated from the functional code, it is sometimes necessary to redefine it completely for all inherited features instead of having local extensions of its parts. These and other difficulties in combining inheritance with concurrency are referred to as *inheritance anomaly* [90].

Thanks to the synchronisation policy based on the new semantics of contracts developed in chapter 8, most inheritance anomalies do not occur in SCOOP; the remaining ones are due to the limitations of the assertion language rather than the chosen synchronisation mechanism. Below, we discuss three inheritance anomalies that most COOLs suffer from: *partitioning of acceptable states*, *modification of acceptable states*, and *history-only sensitiveness of acceptable states*; we show how our framework deals with them. We borrow the examples from the seminal paper by Matsuoka and Yonezawa [90] and a recent survey by Milicia and Sassone [97].

Class *BUFFER* [*G*] in figure 10.27 serves as basis for our discussion. Clients use its features *put* and *get* to store and retrieve elements. (To simplify the comparison with other concurrency approaches, we keep the naming convention used in the above articles; if we followed the strict Eiffel style, *get* would be replaced by a pair of features: *item* and *remove*.) The call validity rule 6.5.3 and the principles of DbC discussed in chapter 8 enforce the proper synchronisation on the client's side. A client may only call a feature of the buffer declared as

```
my_buffer: separate BUFFER [INTEGER]
```

in a context where *my_buffer* is locked on the client's behalf; the client must ensure that the precondition will hold when the feature is applied. This may be achieved by wrapping the calls in routines taking *my_buffer* as actual argument and establishing the required precondition, as illustrated in figure 10.28.

```
class BUFFER [G]
feature
    put (v: G)
            -- Store v.
        require
            not is_full
        do
            ...
        ensure
            not is_empty
        end

    get: G
            -- Remove oldest element and return it .
        require
            not is_empty
        do
            ...
        ensure
            not is_full
        end
    ...
end
```

Figure 10.27: Basic buffer

```
-- in class C
store ( buffer : separate BUFFER [INTEGER]; i: INTEGER)
    require
        not buffer . is_full
    do
        buffer .put (i)
    ensure
        not buffer . is_empty
    end
```

Figure 10.28: Buffer's client

Partitioning of acceptable states

The first common inheritance anomaly is caused by partitioning the states of an object into a number of sets, and enabling the features according to these sets. For instance, the original class *BUFFER* [*G*] has three sets of states identified as *is_empty*, *is_full* , and *is_partial* (neither full nor empty). Assume that we want to introduce a new class *BUFFER2* [*G*] that inherits from *BUFFER* [*G*] and adds a new feature *get_two* which consumes two elements from the buffer (see figure 10.29). Obviously, *get_two* may only be applied when the buffer

holds at least two elements; the set *is_partial* is partitioned into *exactly_one_element* and *at_least_two_elements* .

```
class BUFFER2 [G] inherit BUFFER [G]
feature
    get_two: TUPLE [G, G]
            -- Remove two oldest elements and return them.
        require
            at_least_two_elements
        do
            Result := [get, get]
        ensure
            not is_full
        end
    ...
end
```

Figure 10.29: BUFFER2

In many languages with state-based synchronisation, the new partitioning required by *get_two* forces the redefinition of the inherited features *put* and *get* so that they capture correctly the possible state transitions; hence the inheritance anomaly. SCOOP, which uses state-based synchronisation, would also be prone to this anomaly if preconditions only allowed the use of boolean attributes but not functions or expressions. In that case, the state of a buffer would be captured by attributes *is_empty* and *is_full* ; the bodies of *put* and *get* would perform assignments to these attributes. The introduction of *get_two* would require an addition of the attribute *at_least_two_elements* ; consequently, the inherited *put* and *get* would need to be redefined to set correctly the new attribute. Fortunately, SCOOP supports boolean expressions and functions in preconditions; *is_empty* and *is_full* may be conveniently implemented as functions which do not need updating in routine bodies (see figure 10.30). As a result, the introduction of *get_two* has no impact on the inherited features, i.e. the inheritance anomaly is avoided.

Interestingly, unlike most COOLs, our model enables a straightforward implementation of *get_two*'s functionality without using inheritance at all. It is sufficient for a client to wrap two consecutive calls to *get* in a feature which takes the buffer as argument and lists an appropriate precondition, as ilustrated in figure 10.31. Of course, *count* in class *BUFFER* [*G*] must be exported to *C* in that case.

Modification of acceptable states

The second kind of anomaly arises in the context of multiple inheritance, where a class is "mixed" into another class to enrich its behaviour by adding a feature that influences the acceptance states of the original features. A typical example is the class *LOCKABLE_BUFFER* [*G*] in figure 10.32, obtained through the inheritance from *LOCKABLE* and *BUFFER* [*G*]. One would like the instances of the resulting class to behave just like instances of *BUFFER* [*G*] if *is_locked* is false, but not to accept calls other than *unlock* if *is_locked* is true. In most COOLs, the original version of *put* and *get* would need a redefinition to account for the strengthened synchro-

```
-- in class BUFFER [G]
is_full : BOOLEAN
    do
        Result := count = size
    ensure
        count = size
    end

is_empty: BOOLEAN
    do
        Result := count = 0
    ensure
        count = 0
    end

count: INTEGER
    -- Number of elements

size : INTEGER
    -- Buffer capacity

-- in class BUFFER2 [G]
at_least_two_elements : BOOLEAN
    do
        Result := count >= 2
    end
```

Figure 10.30: Functions used in preconditions

```
-- in class C
get_two (buffer : separate BUFFER [INTEGER]): TUPLE [INTEGER, INTEGER]
        -- Retrieve two elements from buffer .
    require
        buffer .count >= 2
    do
        Result := [ buffer .get , buffer .get ]
    end
```

Figure 10.31: Implementation of *get_two* without inheritance

nisation requirement. SCOOP is immune to this anomaly: *put* and *get* are kept intact; only the functions *is_full* and *is_empty* need to be redefined. Now, *put* and *get* may only be executed if *is_locked* is false because $\neg is\_full \implies \neg is\_locked$ and $\neg is\_empty \implies \neg is\_locked$.

Class *LOCKABLE_BUFFER* [*G*] does not, however, conform to *BUFFER* [*G*]. This is necessary for preserving the compatibility with the rules of DbC. The proposed redefinition of *is_full* and *is_empty* effectively weakens their postconditions and strengthens the precondi-

```
deferred class LOCKABLE
feature
    lock require not is_locked do is_locked := True end
    unlock do is_locked := False end
    is_locked : BOOLEAN
end

class LOCKABLE_BUFFER [G]
inherit {NONE} -- Non-conforming inheritance
    LOCKABLE
    BUFFER [G] redefine is_empty, is_full end
feature
    is_full : BOOLEAN
        do
            Result := is_locked or count = size
        end

    is_empty : BOOLEAN
        do
            Result := is_locked or count = 0
        end
end
```

Figure 10.32: Lockable buffer

tions of *put* and *get*; this is at odds with DbC because clients could be cheated on in the presence of polymorphic calls. Therefore, the *non-conforming inheritance* mechanism [53] must be used for inheriting the features of *BUFFER* [*G*]. (That is why **inherit** {*NONE*} is used instead of the standard **inherit** form.) It breaks the subtyping relation; no polymorphic attachments between the entities of both types are allowed, so there is no risk of incorrect calls. An arbitrary redefinition of contracts is permitted here; if, on the contrary, we used the standard inheritance mechanism, the new versions of *is_full* and *is_empty* could violate their inherited postconditions.

History-only sensitiveness of acceptable states

The third kind of anomaly occurs when the application of a particular feature is enabled or disabled depending on the execution history rather than the current state of the object. Consider the class *HISTORY_BUFFER* [*G*] in figure 10.33; its feature *gget* may only be executed if the previous operation on the buffer was a call to *get*. SCOOP, like many other approaches, does not support history variables; they have to be emulated using "ghost" attributes. This leads to the anomaly: the inherited features need to be redefined to set or reset the corresponding attributes. Here, the new versions of *put* and *get* may rely on their original bodies through the use of **Precursor** calls but this only palliates the problem; we would like to avoid their redefinition altogether.

The anomaly occurs in our model but the problem is not concurrency-specific; it happens in

```
class HISTORY_BUFFER [G]
inherit
    BUFFER [G] redefine put, get end
feature
    put (v: G)
            -- Store v.
        do
            Precursor (v)
            after_get := False
        end

    get: G
        do
            Result := Precursor
            after_get := True
        end

    gget: G
            -- Like get but follows another call to get.
        require
            after_get
            not is_empty
        do
            Result := get
            after_get := False
        end

    after_get : BOOLEAN
            -- History variable
end
```

Figure 10.33: Buffer with history variables

a sequential context as well because the assertion language of Eiffel is not expressive enough to specify a required sequencing of feature calls. Auxiliary attributes used for emulating history variables are themselves prone to the *partitioning of acceptable states* anomaly mentioned earlier.

10.6 Discussion

A number of practical applications of SCOOP have been presented. Several techniques discussed in previous chapters prove essential for achieving compact solutions. Of particular interest are: atomic locking of multiple processors, detachable and attached types, contract-based synchronisation, inheritance, genericity, and asynchronous agent calls (in many different flavours). We have also shown how SCOOP leverages the powerful O-O techniques to provide

extensive support for reuse. Classes written for a sequential application may be readily used in a concurrent one and vice-versa. The model is immune to common inheritance anomalies because contracts allow a clean reuse of synchronisation code: since every inherited feature must respect the original contract, and the synchronisation requirements are part of the contract, the redefined synchronisation code is guaranteed to be compatible with the original. This immediately rules out inheritance patterns that are not compatible with the principles of DbC, as illustrated by the lockable buffer example above. The assertion language is flexible enough to eliminate the anomalies related to the partitioning and the modification of acceptable states. Nevertheless, history-related anomalies persist. They could be eradicated by enriching the contract language but the required extensions are non-trivial.

11

Implementation: issues and solutions

GENTLEMEN never propose things that cannot be implemented. The implementation of SCOOP is a major contribution of this dissertation. It is essential for validating the model and demonstrating its practicality. Our work began with an attempt at implementing SCOOP_97; that initial effort revealed several limitations and inconsistencies of the model, prompting us to redesign it, and ultimately leading to the development of the current SCOOP framework. Several theoretical and practical issues discussed in this dissertation have been uncovered during the implementation work.

SCOOP is supported by three complementary tools:

- $SCOOPLI$ library which implements the basic computational model: processors, atomic locking, scheduling, wait by necessity, and the new contract semantics.

- $scoop2scoopli$ compiler which type-checks SCOOP code following the rules introduced in chapter 6 and translates it into pure Eiffel code with embedded calls to SCOOPLI.

- $CONCURRENCY$ library with a number of utility classes implementing advanced concurrency mechanisms not supported in the basic model. CONCURRENCY classes are written in SCOOP and can be readily used in various applications, e.g. through inheritance, as demonstrated in chapter 10. The library is open; future extensions and modifications, e.g. new facilities for distributed and real-time programming (see section 13.2), are welcome.

Right from the beginning, we decided in favour of a library-based implementation of the basic mechanism. This let us focus on concurrency issues without getting bogged down in the intricacies of existing compilers. Besides, at the outset of this study, no satisfactory open-source Eiffel compiler was available.

Our implementation follows the ECMA/ISO language standard [53, 68] and it is compatible with existing Eiffel tools. Code produced by $scoop2scoopli$ is compliant with the EiffelStudio (versions 5.5–5.7) and Gobo compilers; the tool itself has been implemented on top of the Gobo compiler framework [25]. $Scoop2scoopli$ is a command-line tool but it has been recently integrated with EiffelStudio by Yann Müller [105]. As a result, SCOOP projects can now be created, edited, compiled, and run in a similar way as sequential Eiffel projects; the only notable exception being the lack of debugger support for inspecting individual processors.

11.1 Supported mechanisms

The ultimate goal of the implementation effort is to cover all the mechanisms and techniques supported by the model. Some mechanisms proposed in this dissertation, however, have not been fully implemented. This is due to several reasons, primarily to the fact that some Eiffel extensions introduced in the ECMA/ISO standard have no implementation yet; SCOOP's reliance on this standard has had an impact on the extent of our implementation. This section reviews basic and advanced elements of SCOOP discussed in previous chapters, and comments on the completeness of their implementation.

- *Feature calls*
  Separate and non-separate calls are fully supported. The separateness of a call is determined at run time, following the new semantics of the feature call mechanism (rule 7.2.1).

- *Atomic locking*
  SCOOPLI provides a locking mechanism based on the refined semantics of the feature call mechanism (rule 7.2.1). There is no limit to the number of processors locked atomically.

- *Lock passing*
  The lock passing mechanism introduced in chapter 7 is fully implemented.

- *Fair scheduling*
  The scheduler provided as part of SCOOPLI ensures strong fairness: satisfiable requests are serviced in a FIFO order. This policy is stricter than required by SCOOP; see section 11.2.4.

- *Preconditions, postconditions, and invariants*
  The new semantics of preconditions and postconditions introduced in chapter 8 is fully implemented. Controllability of assertions is detected in two steps: (1) scoop2scoopli performs a static check to filter out all clauses that do not involve separate calls (such clauses are controllable by definition), and (2) controllability or the remaining clauses is checked at run time. Invariant checking is also supported. Contracts are implemented in a way that enables switching on and off run-time assertion monitoring.

- *Loop assertions and **check** instructions*
  The implementation does not follow the new semantics outlined in section 8.1.4; these constructs are treated as in sequential Eiffel, i.e. wait by necessity applies. However, there is no technical obstacle to implementing the new semantics.

- *Attached types*
  The ECMA/ISO specification of attached types has not been implemented in existing Eiffel compilers. Scoop2scoopli supports attached types as far as they influence SCOOP mechanisms, e.g. locking and type checking. Other considerations, e.g. Certified Attachment Patterns and initialisation of attached attributes by creation procedures, are ignored because they fall beyond the scope of this work.

- *Object test*
  Assignment attempts '?=' are used instead of object tests, due to the lack of support for the latter in existing Eiffel compilers.

- *Object comparison*
  Object comparison takes into account the locality of compared objects. This is achieved through an additional comparison of processor identities in feature *is_equal* .

- *Object import*
  The four proposed versions of import operation (see section 6.7) are not implemented. A full implementation of import operations would require support from the runtime. Feature *deep_import* is provided in class *ANY* but it may yield incorrect results for non-separate objects reachable from an imported root object via separate references.

- Separate actual generic parameters are fully supported, but non-separate parameters may behave incorrectly when applied to separate types. Declarations of the form

  l1 : *LIST* [*X*]
  l2 : *LIST* [**separate** *X*]
  l3 : **separate** *LIST* [**separate** *X*]

  are handled correctly, whereas

  l4 : **separate** *LIST* [*X*]

  may lead to inconsistencies because the type of its elements is seen by clients as $(!, \bullet, X)$ rather than the expected $(!, \top, X)$ yielded by the type combinator '$\star$' (see section 9.2).

- *Type rules*
  The type checker of scoop2scoopli uses type rules introduced in chapter 6 to eliminate traitors. Additional conformance rules for generically derived types (rule 9.2.3) have not been implemented yet. At the moment, standard Eiffel type rules apply, e.g. allowing covariant subtyping among actual generic parameters; as a result, it is possible to introduce traitors through an incorrect manipulation of generic types.

- *Enriched type annotations*
  The extended SCOOP syntax is fully supported. Besides the usual '<>' notation for processor tags, an alternative comment-based notation '−−<>' is supported; it allows the use of SCOOP within Eiffel editors which are not (yet) aware of the full SCOOP syntax. For example, type annotations

  y: **separate** <*x*.**handler**> *Y*

  can be rewritten as

  y: **separate** *Y* −−<x.handler>

  Separate annotations may also appear in formal generic parameters, including generic constraints.

- *Agents*
  Agents are supported by SCOOPLI. At the moment, however, scoop2scoopli is unable to produce fully operational code for creation of separate agents. The necessary

manual modifications of generated code are straightforward; we are currently extending scoop2scoopli to eliminate the need for manipulation of produced code.

- *Garbage collection*
  Unused objects are collected automatically using the standard mechanism provided by the Eiffel runtime. Automatic garbage collection of processors has not been implemented. This may pose a problem in applications where a large number of processors are created and used for a short period of time, e.g. with fully asynchronous calls or event-based programming. Manual cleanup is possible; an appropriate feature is provided by SCOOPLI (see section 11.2.7).

The extent to which SCOOP has been implemented is sufficient to demonstrate the feasibility of the approach and apply it in practice. We have found no major technical obstacles that would indicate the infeasibility of a full implementation. On the contrary: the increasing support for new mechanisms and rules of ECMA/ISO Eiffel standard provided by EiffelStudio and Gobo compilers means that a complete implementation of SCOOP is just a matter of time.

11.2 SCOOPLI library

SCOOPLI implements the basic SCOOP abstraction: processors. It also provides support for such essential mechanisms as separate calls, atomic locking of multiple resources, wait semantics of preconditions, wait by necessity, and fair scheduling.

The library is compatible with the EiffelStudio 5.5–5.7 and Gobo compilers. It relies on EiffelBase and EiffelThread libraries, and targets POSIX and NET threading. It can be used on native Windows and .NET 1.1; simple examples have been compiled and tested on Linux but we claim no compatibility with that platform.

SCOOPLI is very compact. It consists of 5 classes representing basic abstractions:

- *SCOOP_PROCESSOR*
- *SCOOP_SCHEDULER*
- *SCOOP_SEPARATE_TYPE*
- *SCOOP_SEPARATE_CLIENT*
- *SCOOP_SEPARATE_PROXY*

and 8 supporting classes; in total, they represent 130 features and 2100 lines of code. Figure 11.1 shows the relations between the most important classes. In the remainder of this section, we show how the essential SCOOP mechanisms are implemented using these classes; we point out the encountered problems, and describe our solutions.

11.2.1 Processors

Processors are instances of *SCOOP_PROCESSOR*. Each processor has an associated thread whose task is to service its *action queue* where feature calls issued by separate clients are stored.

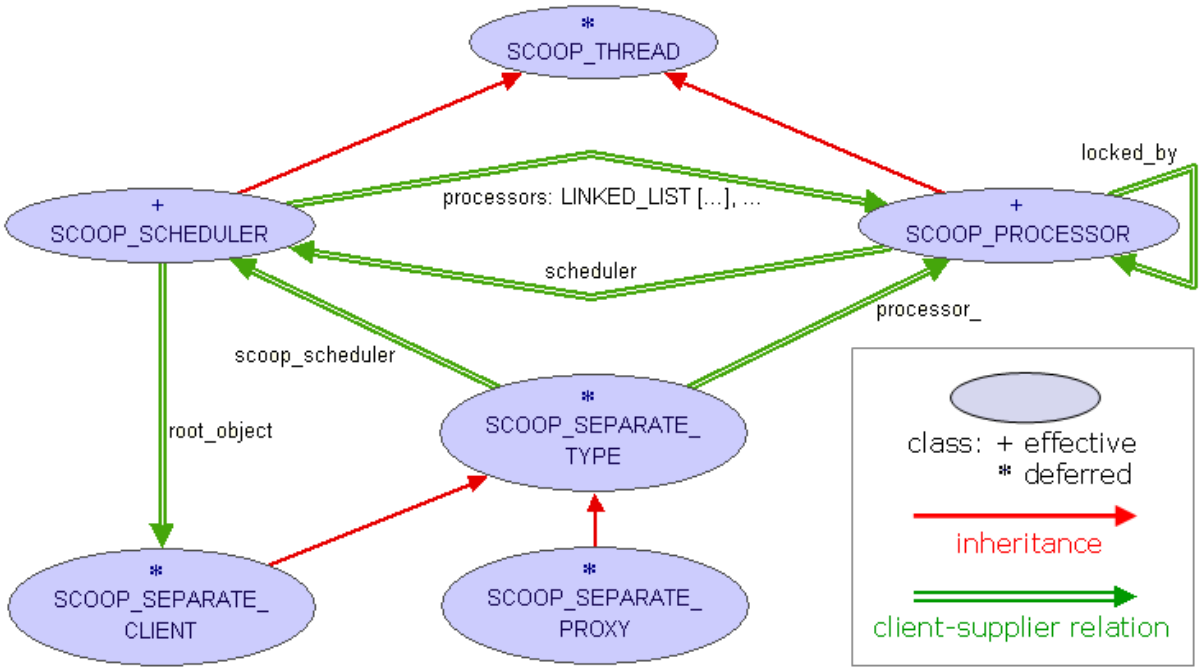


Figure 11.1: SCOOPLI library: basic classes

These requests are executed in the FIFO order. The access to the action queue is protected by a mutex to avoid data races between a client processor extending the queue and the processor handling it. *Wait by necessity* is implemented at the level of the action queue: a client processor adding a query request is blocked until the request has been processed, i.e. the queue has become empty and the query has been executed. [1].

A processor may be locked by another processor, or free. A reference to the lock owner is provided by the query *locked_by*. The lock owner has the right to access the processor's action queue; other processors cannot do it, unless they received the lock as a result of lock passing (see section 11.2.5 below).

Each processor has a reference to the centralised scheduler; on creation, the processor registers with the scheduler by putting itself into the scheduler's list *processor_list* . Processors do not keep references to objects they handle; it is the objects that keep a reference to their handler. This is necessary to support efficient garbage collection (see section 11.2.7).

11.2.2 Separate objects

In a separate call, an object may act as *separate client* or as *separate supplier*. A separate client must know its own processor and have access to the centralised scheduler, so that it can issue lock requests associated with routines taking separate arguments. Class *SCOOP_SEPARATE_CLIENT* provides these facilities in the form of features *processor_* and *separate_execute_procedure* ; application classes that declare separate entities and uses calls to routines with separate arguments must inherit from that class. Class *SCOOP_SEPARATE_PROXY* represents separate suppliers, more precisely proxies to such sup-

[1] A busy wait loop (used initially in YO_SCOOPLI) has now been replaced by a more sophisticated signalling policy based on wait handles implemented by the SCOOPLI class *SCOOP_AUTO_RESET_EVENT*

pliers. The actual supplier objects need not be aware that they participate in separate calls; proxies must take care of the necessary synchronisation, and delegate calls to the actual supplier objects. Therefore, declarations of separate entities, e.g.

```
my_x: separate X
my_Y: separate Y
r (y: separate Y) do ... end
```

must be rewritten as

```
my_x: SCOOP_SEPARATE__X
my_y: SCOOP_SEPARATE__Y
r (y: SCOOP_SEPARATE__Y) do ... end
```

where *SCOOP_SEPARATE__X* and *SCOOP_SEPARATE__Y* are proxies for classes *X* and *Y* respectively. (Proxy classes may be given different names; the above example uses the standard naming scheme applied by scoop2scoopli.)

11.2.3 Separate calls

Calls to routines taking separate formal arguments must be wrapped in agents and passed to the centralised scheduler where a corresponding locking request is created and serviced (see section 11.2.4 below), e.g.

```
r (my_y)
```

becomes

```
scoop_separate_execute ([my_y.processor_ ], agent r(my_y), agent r_wait_condition ,
                        agent r_separate_post , agent r_non_separate_post )
```

This call results in waiting until a lock on *my_y*'s handler is acquired on behalf of the client's processor, and *r*'s uncontrolled preconditions hold (they must be extracted into a wrapper query *r_wait_condition*), then executing *r*'s body and checking its postconditions. The postconditions are extracted into two wrapper queries *r_separate_post* and *r_non_separate_post* . The distinction between separate and non-separate postcondition clauses in necessary for implementing their new semantics; see section 11.3.4.

Each separate command call of the form $y.f$ (...) must be wrapped in a call to *scoop_asynchronous_execute*, e.g.

```
y.f (...)
```

becomes

```
scoop_asynchronous_execute (Current, agent y.f (...) )
```

where the first actual argument indicates the separate client and the second denotes the requested feature call. Similarly, each query call $y.q$ (...) must be wrapped in a call to *scoop_synchronous_execute*; additionally, the result of the query must be retrieved.

Auxiliary features *scoop_asynchronous_execute* and *scoop_synchronous_execute* are provided in class *SCOOP_SEPARATE_PROXY*, so that wrapping can be conveniently done in the proxy class; with this approach, the original calls $y.f$ (...) and $y.q$ (..) need no modifications. (Scoop2scoopli uses this technique, see section 11.3.1 for details.)

11.2.4 Scheduling, locking, and wait conditions

A centralised scheduler, implemented by class *SCOOP_SCHEDULER*, processes locking requests and checks wait conditions. The scheduler manages a global routine request queue where locking requests are stored; a dedicated scheduler thread is used to service the requests. The execution of *separate_execute* by a processor P_r, which corresponds to calling an enclosing routine *r* (*my_x*, *my_y*, ...) in the original code, results in creation of a routine request (an instance of *SCOOP_ROUTINE_REQUEST*) and its addition to the global request queue, provided that P_r does not already hold all the necessary locks (in which case it can proceed immediately). The routine request contains four pieces of information:

- The identity of P_r.
- A tuple of processors to be locked: [P_{my_x}, P_{my_y}, ...].
- An agent representing the wait condition.
- An agent representing the enclosing routine *r*.

After adding its request to the queue, P_c is blocked until the request is satisfied. To avoid starvation, routine requests are considered in a FIFO order; this policy ensures that every satisfiable request is serviced before requests issued later. A request might be in one of three states:

- *idle*, i.e. it is waiting to be serviced.
- *active*, i.e. locking and wait condition checking have been successful; the corresponding enclosing routine *r* is being executed by the requesting processor P_r.
- *finished*, i.e. P_r has terminated the execution of *r* and is ready to relinquish the locks.

If the scheduler comes across an *idle* request, it tries to acquire locks on the requested processors on behalf of the client processor; if locks are acquired, a wait condition check follows. If either operation fails, the scheduler moves to the next request, leaving the current one idle. If both locking and wait condition checking are successful, the request is marked as *active* to give the client processor a permission to execute the body of the associated routine; the client processor will change the state of the request to *finished* as soon as it terminates the routine. The scheduler moves to the next request, and so on. If a request is marked as *finished*, the scheduler unlocks all processors locked by that request, removes it from the queue, and restarts servicing the queue from the head. The latter step is necessary to ensure that no earlier requests which now have become satisfiable (due to the last unlocking operation) are omitted.

This scheduling policy is stricter than required for implementing the fairness guarantees of SCOOP. We have decided to use it because of its simplicity and low run-time overhead: only one scheduler thread is necessary. In some situations, however, this solution is suboptimal. For example, two requests with disjoint sets of requested processors are serviced one after the other, although they could be taken care of in parallel. On the other hand, our scheduler is immune to starvation, unlike Compton's lock manager (see section 4.3).

11.2.5 Lock passing

For simple locking, as in SCOOP_97, a single query *locked_by* in class *SCOOP_PROCESSOR* would suffice. To support lock passing, however, every processor must additionally keep track of all the processors it has locked. The scheduler makes use of this information to check whether a client processor P_c requesting a lock on a supplier processor P_x already owns this lock as a result of (a chain of) lock passing operations. Conversely, every processor must know the set of all processors that (indirectly) hold a lock on it (*locked_by* only denotes a direct lock holder). This information is needed by the client processor P_c when issuing a separate call on the supplier processor P_x. P_c must check whether P_x holds a lock on P_c, which would indicate that the call is a separate callback and should be processed synchronously rather than being added to the action queue of P_x, as implied by the rule 7.2.1. Class *SCOOP_PROCESSOR* has been therefore extended with features *locked_processors* and *synchronous_processors*. Both of them represent stacks of processors and always contain at least one element: the current processor itself. Before a feature call involving lock passing the client processor pushes the contents of its *locked_processors* stack on the supplier processor's; the same operation is performed for *synchronous_processors*. After the feature application (lock passing implies waiting, see rule 7.2.1) the supplier processor's stacks are chopped off to their previous size. (Scoop2scoopli automatically inserts the appropriate calls before and after the feature application.) The stack-based implementation of processor sets permits correct handling of recursive lock passing and separate callbacks. The run-time deadlock detection mechanism for SCOOPLI implemented by Daniel Moser [100] also makes use of *locked_processors* stacks.

11.2.6 Quiescence and termination

The scheduler performs a quiescence detection step every time the request queue becomes empty and no processors have reported activity. The quiescence detection consists in checking that all processors are indeed idle (processors only report activity when an action terminates, so it is possible that some processor are busy handling their action queues). If yes, the whole system is terminated: the scheduler asks processors to exit their action loops and remove themselves from *processor_list* ; as soon as the list is empty, the scheduler terminates its loop. This marks the end of the execution.

Quiescence detection introduces little run-time overhead because it is only performed when the scheduler thread has other activity, i.e. no requests to service.

11.2.7 Garbage collection

If an object is not referenced by other objects on the same processor (a direct reference) or on another processor (through a proxy), it can be garbage-collected. As soon as a proxy is not referenced by any other object, it can be garbage-collected too. Processors do not keep references to the object structures they handle, so the standard garbage collection mechanism is sufficient.

Processor objects, however, are not collected automatically. Even if no application object in the system references a given processor, the single reference kept by the scheduler (in the processor list) prevents the garbage collector from doing its work. In most applications with a bounded number of processors it poses no particular problems. The advanced agent-based

mechanisms implemented in the CONCURRENCY library, however, often create a large number of auxiliary processors which are only used for single calls and then left idle. For example, every application of *asynch* creates an additional processor; every use of *parallel_or* with n targets creates n+1 processors, and so on. To avoid multiplication of idle processors, manual garbage collection is possible in such cases. Class *SCOOP_SCHEDULER* provides a feature *remove_processor* which deletes the entry for a given processor from the list of processors kept in the scheduler. As soon as that last reference disappears, the automatic garbage collector can kick in and dispose of the processor.

Discussion

SCOOPLI maps SCOOP concepts to the underlying multithreading platform in a straightforward and faithful way: all actions on a given object are executed by a thread representing the object's handler. Furthermore, the scheduler thread does not apply any features on behalf of a client; it only notifies the client that the enclosing routine associated with a request can be executed; the execution of the routine is performed by the client's processor. Separate calls are handled exactly as prescribed by the feature call semantics (rule 7.2.1), i.e. instead of executing a given feature directly, the client's processor asks the supplier's processor to execute it. Lock passing obeys the same basic rule: features are always executed by a thread representing the supplier's processor, never by the client's own thread.

Our implementation uses threads very sparingly: there is one thread per processor, plus one scheduler thread; in total n+1 threads for a system composed of n processors. Despite the relatively high run-time overhead due to agent wrapping of separate calls, we have chosen this implementation approach for three important reasons:

- The run-time model is very close to the abstract operational semantics; this facilitates the understanding of the practical impact of formal rules introduced in the model.
- Keeping the basic mechanisms at a high level of abstraction should facilitate future implementations. Large parts of the library can be reused without modification in implementations targeting other platforms, e.g. for distributed computing. SCOOPLI relies on threading but this binding is weak; other execution facility can easily replace threads.
- The main purpose of this implementation effort is to demonstrate the feasibility and practicality of the SCOOP model; efficiency is considered but it is less important.

11.3 Scoop2scoopli tool

SCOOPLI provides all the basic concurrency mechanisms but its direct use would be burdensome because the necessary wrapping of separate calls, creation of proxies, and explicit calls to the scheduler obscure the syntax; manual manipulation of synchronisation features is error-prone. Therefore, we have implemented the scoop2scoopli tool which translates SCOOP programs into Eiffel code with embedded calls to SCOOPLI features, so that programmers need not deal directly with the library. The tool was first conceived as a pre-processor but later a type-checker was added; therefore, we view it as a SCOOP-to-Eiffel "compiler".

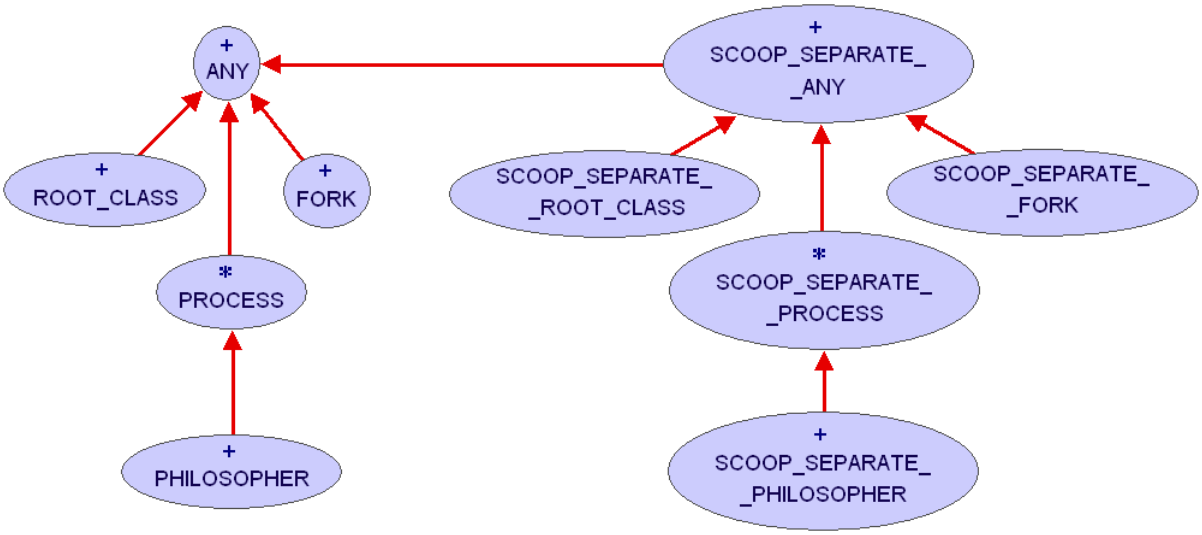


Figure 11.2: Example class hierarchy generated by scoop2scoopli

Compilation of SCOOP projects

SCOOP projects can be compiled from the command line be executing the command

`scoop2scoopli` *filename.ace*

where *filename.ace* is the name of a project configuration file in the ACE format supported by Eiffel compilers. The configuration file must specify a system name, a root class and a root creation procedure, and a list of clusters with SCOOP classes.

Scoop2scoopli type-checks the program and creates a subdirectory *scoop_build* where a new project configuration file (the name of the configuration file corresponds to the specified system name) and generated Eiffel classes are placed. The generated classes are organised in directories following the original cluster hierarchy. The configuration file produced by scoop2scoopli can now be submitted to the EiffelStudio compiler to obtain an executable from the generated classes.

Optional compilation switches $\backslash nolockpassing$ and $\backslash seqpost$ may be used with scoop2scoopli to simulate SCOOP_97 semantics: $\backslash nolockpassing$ turns off the lock passing mechanism; $\backslash seqpost$ enforces the sequential semantics of postconditions.

11.3.1 Code generation

Two classes are generated for each SCOOP class *C*: an original class called *C* and a proxy class called *SCOOP_SEPARATE__C*. Figure 11.2 shows the class hierarchy for the dining philosophers example described in section 10.1.1. (The figure only includes the relevant classes; libraries are omitted.) The original classes (the left-hand side of the diagram) correspond to those of the compiled SCOOP system; SCOOP code, however, has been replaced by pure Eiffel code with embedded calls to SCOOPLI. The proxy classes (the right-hand side of the diagram) are proxies to the original classes; their hierarchy mirrors the original one.

Generated classes are put into two subdirectories of *scoop_build*. Additionally, root class *SCOOP_STARTER* is generated and put in *scoop_build*. The new project configuration file specifies {*SCOOP_STARTER*}.*execute* as root creation procedure of the produced Eiffel system. See section 11.3.2 for details concerning *SCOOP_STARTER* and bootstrapping of SCOOP applications.

Original classes

Each SCOOP class in the compiled system is transformed into an Eiffel class. SCOOP constructs are replaced with appropriate calls to SCOOPLI features; a few more modifications are necessary.

- Every class inherits from *SCOOP_SEPARATE_CLIENT* (see section 11.2.3); inheritance links are automatically inserted by scoop2scoopli.
- Each declaration of a separate entity is turned into a declaration using the corresponding proxy class, e.g.

  x: **separate** X

  becomes

  x: *SCOOP_SEPARATE__X*

- Separate calls take one additional argument 'Current', e.g.

  $x.f$ (5, ''*Hello World*!'')

  becomes

  $x.f$ (**Current**, 5, ''*Hello World*!'')

- Since separate calls now take one more argument, infix features on separate targets cannot be used. Calls to infix features are replaced with appropriate calls to their non-infix synonyms

  if $x > y$ **then** ... **end**

  becomes

  if x. *non_infix_gt* (**Current**, y) **then** ... **end**

- Every routine taking separate formal arguments is split into three routines — one issuing a locking request to the scheduler through *separate_execute_routine* , one wrapping the body, and one wrapping the wait conditions — following the technique outlined in section 11.2.3.
- Creation of separate objects is split in two steps: (1) creation of a proxy with an associated processor, and (2) calling the creation procedure, e.g.

  create $x.make$

  becomes

```
create x. set_processor_ (scoop_scheduler.new_processor_)
separate_execute_procedure ([x], agent x.make (Current), Void)
```

- Each assignment attempt is followed by an additional **if** ... **then** ... **else** statement checking the conformance of processors between the source and the target.

The rest of the code is left unchanged; as a result, the generated classes are very similar to the original ones. In particular, the original inheritance hierarchy is preserved (except for adding a link to *SCOOP_SEPARATE_CLIENT*).

Proxy classes

Proxy classes implement separate types. Their hierarchy follows that of the original classes, e.g. if class *C* inherits from *A* and *B* then the proxy *SCOOP_SEPARATE__C* inherits from *SCOOP_SEPARATE__A* and *SCOOP_SEPARATE__B*; feature renaming, redefinition, and merging is taken into account. No inheritance relation exists between *SCOOP_SEPARATE__C* and *C*; their conformance is enforced through *conversion*; an appropriate procedure is introduced in the proxy class. Proxies use delegation to access features of original classes. For that purpose, *SCOOP_SEPARATE__C* declares an attribute

```
implementation_: C
```

representing the actual object to which feature calls are redirected. To cater for various mechanisms of SCOOP — separate calls, locking, lock passing, and object creation — redirected calls must be wrapped in calls to appropriate SCOOPLI features.

- For each command *f* (...) from class *C*, *SCOOP_SEPARATE__CLIENT* lists a routine

```
f (a_caller: SCOOP_SEPARATE_CLIENT; ...)
    do
        ... -- Check each argument of proxy type represents actual object;
            -- if not, use Void in the actual call.
        separate_execute (a_caller_, agent implementation_.f (...)
            -- Actual call
    end
end
```

- A similar routine is provided for each query *q* (...) : *X* but it additionally retrieves the result of the actual call.

```
q (a_caller_: SCOOP_SEPARATE_TYPE; ...): SCOOP_SEPARATE__X
    local
        a_function_to_evaluate: FUNCTION [ANY, TUPLE, X]
    do
        a_function_to_evaluate := agent implementation_.q (...)
        scoop_synchronous_execute (a_caller_, a_function_to_evaluate)
        Result := a_function_to_evaluate.last_result
    end
```

- Since agent creation on attributes, constants, and once functions is not supported by the EiffelStudio compiler, wrappers are created for such features; these wrappers are used in calls to *scoop_separate_execute* .
- For each creation procedure *make* (...) three features are generated in the proxy class

```
make ( a_caller_ : SCOOP_SEPARATE_CLIENT; ...)
    do
        separate_execute ( a_caller_ , agent implementation_.make (...) )
    end

make_scoop_separate__c ( a_caller_ : SCOOP_SEPARATE_CLIENT; ...)
    do
        separate_execute ( a_caller_ , agent
            effective_make_scoop_separate__c (n))
    end

effective_make_scoop_separate__c (...)
    do
        create implementation_.make (n)
    end
```

  The first feature is used in non-creation calls to *make*; the latter two in creation calls. This is necessary to cater for the changing creation status of *make* along the inheritance hierarchy, and for a correct handling of dynamic binding in the presence of redefinition and renaming (note that the two wrapper features are suffixed with the class name).
- Routines taking separate arguments check whether lock passing should occur before delegating the call to the actual object. If lock passing is needed, a transfer of *locked_processors* and *synchronous_processors* between the caller's processor and the actual object's processor takes place. The actual call is then scheduled and the the proxy blocks until it has terminated. Finally, locks are revoked.

All proxy classes inherit from *SCOOP_SEPARATE_PROXY* which provides such essential features as references to the handling processor and the centralised scheduler, and the routine *scoop_separate_execute* .

11.3.2 Bootstrapping

The bootstrapping process SCOOP applications is slightly more complicated than for sequential programs. This is due to the necessity of locking the processor that executes the original root creation procedure. Otherwise, the root creation procedure would be executed without its handler being locked. This could result in a violation of mutual exclusion if a reference to **Current** or another non-separate object was passed to a separate supplier; that supplier would be able to acquire a lock on the root processor and perform a callback without waiting for the root creation procedure to terminate.

In the dining philosopher application depicted in figure 11.2, the original system specifies {*ROOT_CLASS*}.*make* as root creation procedure. The code generated by scoop2scoopli, however, starts off the scheduler before creating an instance of *ROOT_CLASS* and calling *make* on

it. That instance of *ROOT_CLASS* must have an associated handler; following SCOOP rules, calling *make* requires a lock on that handler. This is achieved by specifying *SCOOP_STARTER* as root class of the final system. Class *SCOOP_STARTER* is automatically generated by scoop2scoopli; it declares an attribute of type **separate** *ROOT_CLASS*, creates a corresponding object, and issues a call to its feature *make*. The latter call obeys the usual rules, i.e. an enclosing routine is used. This ensures correct locking of the actual root object and prevents other processors from locking its processor before *make* has terminated.

11.3.3 Invariant checking

Spurious invariant violations could when using a creation procedure that takes separate formal arguments. This is due to the implementation of such routines: they do not execute the body of the original creation routine but only call *separate_execute_routine* , passing to it an agent representing the actual body. A call on that agent, performed by the current processor after acquiring the necessary locks and checking the preconditions, catches **Current** in a state where the invariant may not hold. Of course, the invariant should be checked after the execution of the creation routine's body, not before; therefore, invariants are disabled temporarily and reenabled at the end of the creation routine. This requires a transformation of invariant clauses into implications, e.g.

```
invariant
    left_fork  /= Void
    right_fork  /= Void
```

becomes

```
invariant
    (not  invariant_disabled ) implies ( left_fork  /= Void)
    (not  invariant_disabled ) implies ( right_fork  /= Void)
```

and inserting the appropriate assignments to *invariant_disabled* in all concerned creation procedures, e.g.

```
make ( a_left_fork , a_right_fork : SCOOP_SEPARATE__FORK)
    require
        forks_exist : a_left_fork /= Void and then a_right_fork /= Void
    do
        invariant_disabled := True        -- disable invariants
        separate_execute_routine (...)
        invariant_disabled := False       -- enable invariants
    end
```

11.3.4 Postcondition checking

To support the new semantics of postconditions (see section 8.1.2), routines taking separate formal arguments must be transformed so that all postcondition clauses that not involving non-separate calls are extracted and placed in an auxiliary routine to be evaluated asynchronously. For example, an original routine

```
spawn_two_activities (l1, l2: separate LOCATION)
        -- Spawn jobs at l1 and l2.
    do
        ...
    ensure
        post_1: l1.ready
        post_2: l2.ready
        post_3: non_separate_property and then l1.some_property
    end
```

only keeps its last postcondition clause

```
post_3: non_separate_property and then l1.some_property
```

whereas the other clauses are put into individual wrapper routines which are passed as agents to the processors involved in the evaluation of the clauses. In the above example, *post_1* is passed to *l1*'s processor, whereas *post_2* is passed to *l2*'s processor. In cases where several processors are involved in the same clause, the processor handling the first target (in the order of appearance in the formal argument list) is asked to evaluate the clause; it takes care of asking the other processor to evaluate their subclauses. This algorithm avoids cross-locking of the involved processors and is therefore deadlock-free.

An interesting point of this solution is that the evaluation of individual wrappers is itself placed in a *postcondition* of an auxiliary routine, so that run-time postcondition monitoring can be switched off like in sequential systems.

Discussion

Yann Müller has integrated the tool with EiffelStudio 5.7 as part of his MSc project [105]. The additional machinery needed to implement concurrency is hidden from EiffelStudio users. Programmers manipulate SCOOP source code using the built-in editor but preprocessed code is run when an application is launched. Compiling a SCOOP application involves full type-checking and Eiffel code generation with scoop2scoopli, followed by a standard compilation process of the generated code (including SCOOPLI and EiffelThread libraries). The *Melting Ice* technology of EiffelStudio is supported: scoop2scoopli only updates classes that have been modified, so that recompilation is incremental.

Müller's work brings SCOOP programming experience to a new level: projects can now be created, edited, compiled, and run in a similar way as with sequential Eiffel. Unfortunately, the lack of debugger support for inspecting individual processors complicates this task.

11.4 CONCURRENCY library

The CONCURRENCY library provides a set of utility classes supporting advanced concurrency features; programmers may use these classes directly in their code, e.g. through inheritance. The following mechanisms are implemented:

- *call*: universal enclosing routine.

- *asynch*: fully asynchronous calls.
- *waiting faster*: parallel wait on multiple queries.
- *resource pooling*: calling *m* out of *n* separate targets.
- *SCOOP-enabled event types*: asynchronous publication of events and independent notification of multiple subscribed objects.

Several examples in section 10.2 illustrate the use of these facilities. This section reviews the library classes and their essential features. Appendix A includes the full code of all provided classes.

11.4.1 *CONCURRENCY*

The deferred class *CONCURRENCY* provides an interface for the above library facilities. To use these facilities, application classes must inherit from *CONCURRENCY*. The essential features are

- *call* (*a_feature* : **separate** *ROUTINE* [*ANY*, *TUPLE*])
  A universal enclosing routine which can be used for wrapping single separate calls. It takes an agent as argument, e.g.

  call (**agent** *x*.*f* (...))

  See section 9.3.4 for more details.

- *asynch* (*a_feature* : ?**separate** *ROUTINE* [*ANY*, *TUPLE*])
  Performs a fully asynchronous call to the agent passed as argument, e.g.

  asynch (**agent** *x*.*f* (...))

  Relies on class *EXECUTOR*. See section 9.3.4 for a detailed discussion and examples.

- *evaluated_in_parallel* (*a_queries* : *LIST* [?**separate** *FUNCTION* [*ANY*, *TUPLE*, ?**separate** *ANY*]];
  an_initial_answer , *a_ready_answer*: ?**separate** *ANY*;
  an_operator: *FUNCTION* [*ANY*, *TUPLE*, ?**separate** *ANY*]): ?**separate** *ANY*

  Evaluates in parallel a set of separate queries and returns a result as soon as possible. Relies on class *ANSWER_COLLECTOR*. Results of individual queries are combined using *an_operator*. Three operators are defined in class *CONCURRENCY*: *parallel_or* , *parallel_and* , and *parallel_sum* .

- *call_m_out_of_n* (*a_feature* : *ROUTINE* [*ANY*, *TUPLE*];
  a_pool: *LIST* [?**separate** *ANY*]];
  m: *INTEGER*)

  Applies *a_feature* to *m* targets from *a_pool*. Relies on classes *POOL_MANAGER* and *LOCKER*.

11.4.2 *EXECUTOR*

Class *EXECUTOR* is used by *asynch* and *resource pooling* mechanisms; it also serves as ancestor for *EVALUATOR* used in *parallel wait* and for *LOCKER* used in *call_m_out_of_n* . The essential features are

- *execute* (*a_feature* : ?**separate** *ROUTINE* [*ANY*, *TUPLE*])
  Applies *a_feature* after locking its target. Used by {*CONCURRENCY*}.*asynch*.

- *apply_to_target* (*a_feature* : **separate** *ROUTINE* [*ANY*, *TUPLE*];
  a_target : *ANY*)

  Applies an imported copy of *a_feature* to *a_target* . Used by *POOL_MANAGER*.

11.4.3 *ANSWER_COLLECTOR*

Class *ANSWER_COLLECTOR* is used by the *parallel wait* mechanism; it gathers individual results and combines them into a final answer given to a client. The essential features are

- *answer*: ?**separate** *ANY*
  Current answer. Becomes a valid answer, i.e. can be given to the client, if *is_ready* holds.

- *update_answer* (*a_result* : ?**separate** *ANY*)
  Updates *answer* with the individual result *a_result* submitted by an instance of *EVALUATOR*.

- *combinator*: *FUNCTION* [*ANY*, *TUPLE*, ?**separate** *ANY*]
  Used by *update_answer* to combine individual results.

- *ready_answer*: ?**separate** *ANY*
  Value of combined result that allows ignoring any further individual results. For example, *parallel_or* sets it to **True** because a disjunction of boolean expressions is true as soon as one of individual expressions is true.

- *is_ready* : *BOOLEAN*
  Indicates whether *answer* is valid. Set to **True** if *answer* = *ready_aswer* or if all evaluators have submitted individual results.

11.4.4 *EVALUATOR*

Class *EVALUATOR* implements asynchronous evaluation of individual queries and their reporting to *ANSWER_COLLECTOR*. It is used by the *parallel wait* mechanism. The essential features are

- *evaluate_and_report* (*a_query*: **separate** *FUNCTION*
  [*ANY*, *TUPLE*, ?**separate** *ANY*])

  Evaluates *a_query* and reports its result using *notify_answer_collector* .

- *notify_answer_collector* (*an_answer_collector* : **separate** *ANSWER_COLLECTOR*;
  a_result : ?**separate** *ANY*)

  Reports an individual result *a_result* to *an_answer_collector* .

11.4.5 *POOL_MANAGER*

Class *POOL_MANAGER* supports the resource pooling mechanism accessible through *call_m_out_of_n* in class *CONCURRENCY*. The essential features are

- *try_to_apply_feature* (*a_target* : **separate** *ANY*)
  Applies *feature_to_apply* to *a_target* if less than *m* targets have been already used; it does nothing otherwise.
- *feature_to_apply* : *ROUTINE* [*ANY*, *TUPLE*]
  Feature to be applied on *m* targets.
- *m*: *INTEGER*
  Number of requested resources.

11.4.6 *LOCKER*

Class *LOCKER* is used by the resource pooling mechanism *call_m_out_of_n* . One instance of *LOCKER* is created for each element of the resource pool. The essential features are

- *lock_target_and_report* (*a_target* : **separate** *ANY*)
  Locks *a_target* and reports it to *pool_manager* using *report* .

  report (*a_pool_manager*: **separate** *POOL_MANAGER*;
  a_target : **separate** *ANY*)

  Passes locks on *a_target* to *a_pool_manager* and asks the latter to apply the requested feature (through *try_to_apply_feature*).
- *pool_manager*: **separate** *POOL_MANAGER*
  Manager of the resource pool.

11.4.7 SCOOP-enabled *EVENT_TYPE*

This is a SCOOP-enabled version of *EVENT_TYPE*. The original class, proposed by Arslan et al. [12], provided synchronous event publication and notification of subscribed objects. The new version supports fully asynchronous semantics of both operations; see section 10.2.4 for a detailed discussion and examples. The essential features of *EVENT_TYPE* are

- *subscribe* (*an_action* : ?**separate** *ROUTINE* [*ANY*, *EVENT_DATA*])
  Add *an_action* to the subscription list.
- *unsubscribe* (*an_action* : ?**separate** *ROUTINE* [*ANY*, *EVENT_DATA*])
  Remove *an_action* from the subscription list.

- *publish* (*arguments*: *EVENT_DATA*)
  Notify all actions in the subscription list.

There is no additional feature to represent the subscription list because each instance of *EVENT_TYPE* is itself a list; this is achieved by inheriting from *LINKED_LIST*.

Discussion

The main purpose of the library is to provide a repository of useful mechanisms that are not directly supported by the basic SCOOP model. The library is open: the existing facilities can be easily extended, e.g. by introducing new parallel operators in the class *CONCURRENCY*, or by adding new classes and features to implement advanced mechanisms for real-time programming, as suggested in section 13.2. An exception handling mechanism for SCOOP could also rely (at least partly) on library classes.

12

Teaching SCOOP

THE practicality of a programming method depends, in the first place, on its simplicity; tool support is important but even the best tools cannot help if the method is difficult to learn. We taught SCOOP in two iterations of a graduate course at the ETH Zurich. No structured study of students' performance was conducted, but the course participants were asked to provide feedback on SCOOP and its tools, which allowed an informal assessment of SCOOP's usability and ease of learning. This chapter reports shortly on this experience: we discuss topics covered in the course, final project development, and students' feedback.

SCOOP was taught as part of the Concurrent Programming II course[1], henceforth referred to as COOP (*Concurrent Object-Oriented Programming*). The course participants had various backgrounds in terms of industrial experience and previous use of concurrency mechanisms but none of them was familiar with SCOOP. There was an important gap between the participants in terms of DbC and Eiffel experience: the 2005 group had no previous experience whereas the 2006 group had already learnt Eiffel in their first semester of study at the ETH. An introduction to DbC and Eiffel was necessary for the first group, which reduced the number of lectures and exercise sessions dedicated to SCOOP. In 2006, on the other hand, we were able to cover most aspects of the model. Additionally, students were asked for targeted feedback on SCOOP. For these reasons, the rest of the chapter refers to the second iteration of the course. There were 30 participants in that iteration: 22 students (between the 6^{th} and the 9^{th} semester of their CS studies) and 8 industry participants.

The teaching material developed for the course — lecture slides, exercise sheets, and examples — is available online on the course page

`http://se.ethz.ch/teaching/ss2006/0268`

12.1 Topics

The official name of the course is a bit misleading: it suggests that students are required to have taken Concurrent Programming I; that course, however, is not a prerequisite. Therefore, COOP included an introductory part with a historical overview of concurrency including classical and O-O techniques, from semaphores and monitors to active objects to multithreading with Java and C♯. The main part of the course was dedicated to SCOOP. We taught the basics of SCOOP, the application of DbC to concurrent systems, and the use of advanced mechanisms: lock passing, object tests, genericity, and agents. SCOOP was compared with other models:

[1] course number 251-0268-00

Java multithreading, Ada tasking, and active objects. The accompanying exercises proved essential in letting students get acquainted with the practice of concurrent programming, using our framework to solve several concurrency problems before tackling a more challenging final project, and assessing SCOOP with respect to other approaches. The following topics were covered:

- Foundations of concurrency: parallelism, distribution, tasks, threads, preemption, scheduling.
- Classical approaches: mutex, semaphore, monitor, conditional critical region.
- Classical approaches: rendezvous, Ada tasking.
- Classical approaches: actors, active objects, inheritance anomaly.
- SCOOP: processors, separate calls, synchronisation.
- SCOOP: traitors, validity rules, type system.
- SCOOP: Design by Contract in a concurrent setting.
- SCOOP: object import, lock passing.
- Advanced SCOOP techniques: towards real-time programming, events.
- Advanced SCOOP techniques: inheritance, polymorphism, once functions.
- Advanced SCOOP techniques: genericity, agents.

12.2 Assessment

Students were assessed based on a final project and a written exam. The project accounted for 65% of the final mark; the exam represented 35%. We decided to separate the topics: the exam only included non-SCOOP topics, whereas the final project was entirely SCOOP-based. It consisted of two parts:

- Implementation of the classical Santa Claus scenario [140]. The main purpose of this part was to use the basic SCOOP mechanisms to provide a variety of synchronisation pattern present in the example: mutual exclusion, barrier synchronisation, priority scheduling. (We have discussed this example in detail and provided a model solution in section 10.1.4.)
- Implementation of a modified Santa Claus scenario based on active objects (see the project sheet on the course web page for the exact specification of the problem). The purpose of this part was to discover the limitations of SCOOP and emulate synchronisation patterns not supported directly by the basic model. The *asynch* mechanism, which enables fully asynchronous calls and enables simple simulation of active objects as outlined in section 10.2.1, was not implemented yet; students had to come up with their own patterns for representing active objects in SCOOP.

12.3 Students' feedback

The overall quality of the course and the exercise sessions was assessed on the basis of a standard ETH questionnaire. The results of that survey (in German) are available on the course website. Additionally, we asked four questions targeting specifically the SCOOP model and its tools:

- As a programmer, what is your feeling about SCOOP? Does it allow you to program efficiently and express what you want to express?
- How would you compare SCOOP with other mechanisms (say Java multithreading with monitors) in terms of expressivity, usability and "programmer-friendliness"?
- Can you think of additional mechanisms or abstractions that SCOOP should offer?
- What is your opinion on scoop2scoopli? How would you improve the tool?

Overall, students' answers indicated that SCOOP's reliance on O-O principles and Design by Contract facilitated its understanding and practical use. The simplicity of the necessary language extension was also appreciated, as it directly translated into ease of use. Grasping the basic concepts was simple but their application proved more difficult. As pointed out by one of the students, "it takes some time to start thinking the SCOOP way". This effect was more common among the industry participants and students with considerable experience in multithreading; certain reluctance to high-level synchronisation mechanism based on argument passing and contracts could be observed. Students with little or no concurrent programming experience found it easier to adopt the technique.

Most problems experienced during the course and the development of the project were related to the tools. We have to note, however, that the tools were not integrated with EiffelStudio at that time; scoop2scoopli had to be used from the command line, in a combination with the final compiler. In practice, students worked with two instances of EiffelStudio: one for editing the original code, and one for compiling and executing the preprocessed code. This clearly increased the programming burden. Furthermore, the version of scoop2scoopli used in the course did not support full type checking.

Below are the collected answers (quoted directly; each number corresponds to a different project team).

1. *SCOOP model*
    - In our opinion an important aspect of SCOOP, in comparison with other mechanism (Java threading, ...) is that it forces programmers to think more in an object-oriented way. In effect with Java most of the time the concurrent programming style is more low-level, or it is mostly based on procedure that on objects themselves. With SCOOP instead one has always to deal with objects that are the very unit of concurrency. Also debugging is more object-oriented because if something doesn't works as expected you have to reason about which object is waiting for which object, making also debugging more intuitive. Another important aspect is that SCOOP allow to use the full power of Object-Oriented programming and Design by Contract. For example we can use inheritance and polymorphism as we usually do for non-concurrent applications.

- The fact that locking only applies to arguments of routines sometimes decreases the readability of the code since there can be the need of locking an object just to perform a query or a command on it. So we have to introduce a new routine that takes as arguments the objects to lock and maybe just executes one line of code (the call of the command or query on the locked object). This can be gladly accepted if one wants lock the object but if one wants to make a query on one object for which one doesn't need a lock (suppose we want to ask the name of a creature and we know that this name cannot be changed) it can be a little frustrating to always write a new routine with the target of the separate call as argument.
- Locking is sometimes needed at different levels: in our example we sometimes need to lock the rendezvous and sometimes the requests in the rendezvous. Since SCOOP allows us to lock easily at different levels we can ignore the locking of the rendezvous when we only wait for a request to be processed, since we can wait on the request. In this way other rendezvous user can access the rendezvous and submit or process requests.

Additional mechanisms or abstractions that SCOOP should offer

- One possible improvement of SCOOP could be based on the following observation: there is no way in which wait conditions can be "broken" and the processor can execute another routine on the same objects bypassing the block imposed by the preconditions. In SCOOP a processor could be blocked for a big amount of time waiting that a condition becomes true. By taking the example of Ada, we could have a set of open alternatives, try to have them accepted, or eventually have a *delay* t statement that, in case of no request in time t, goes on with other tasks. In this way we avoid letting a processor waiting for a resource (it only checks later if it is become available) and allow it to do some other work in the meantime.

Tools

- Scoop2scoopli is a really simple tool, however having to open two "different" projects in order to test the code with the use of scoop is a bit tedious. The simplicity of creating concurrent programs using scoop which only requires an additional keyword to Eiffel, namely **separate**, is very comfortable for the programmer. However when we look at the interpreted code with all SCOOP calls we see the well hidden complexity behind it. This results sometimes in difficult debugging, and unclear error reports.

2. *SCOOP model*

- Working with SCOOP was an experience, and an enjoyable one. For programmers used to threads and the classic synchronisation tools: mutexes, semaphores, monitors, condition variables, signals, and so on, the change of paradigm is a challenge. But after some practical experience, one gets the grasp on separate entities, processors and lock passing, and from then on, working with SCOOP is a pleasure.

3. *SCOOP model*

   - A lot of the mechanism, like the locking of separate arguments, is implicit. This hides a lot of complexity. It is also quite powerful, because it allows locking of multiple objects "atomically". In comparison with other mechanisms, it often seems to necessitate additional separate communication objects, that explicitly model structures which would have been implicit or passive data structures in other approaches. At first this can be confusing and frustrating, but it seems to lead to relatively nice and elegant solutions. In the frequent, simple case that we need to call a single feature on a separate object, it seems tedious to explicitly have to write a feature that takes the object as an argument and calls the feature. We assume using agents solves this issue nicely.
   - We have found it quite straightforward to reuse code by implementing a lot of common functionality in some base classes that we can inherit from to implement specific behaviour. Even though the mechanisms of SCOOP are quite simple, programming with it has turned out to be more complex than one might think at the beginning.

   Tools

   - Additional mechanisms should be implemented:
     - Support for simplified calling of a single feature on a separate object: *call* (**agent** $a.f$).
     - Support for creating separate objects on a specified processor.

4. *SCOOP model*

   - Efficiency of programming
     One big advantage of programming in SCOOP, is that the programmer doesn't need to remember merely one keyword, **separate**. A drawback, which gets very annoying in time, is the external pre-processor and the recompilation to C every now and then. Especially when tweaking the code, optimizing or debugging it, its a real nuisance to wait for several minutes until you can test your changes. Another disadvantage is the Call Rule, which forces the designers to write tons of wrappers, that let the code quickly explode.
   - Expressive power
     This is great about SCOOP. The Call Rule in combination with wait conditions turns synchronization issues into a piece of cake (slightly exaggerated, but you get the point). So this ruling isn't all negative. Also a big plus is the possibility not to only attach the **separate** keyword to classes but to each instance on its own. This lets the programmer feel much more powerful, if he knows "what's running where". This hopefully gets even better, when the SCOOP type system is going to be fully integrated.
   - Comparison with Java
     As usual, Java lets you just code what you want, without thoroughly thinking about it. This is clearly forbidden in SCOOP (or Eiffel in general), where you are forced

by the language to apply certain patterns, e.g. the Call Rule as mentioned before. This can be a pro or a con, depending on the point of view. In Java, you have to decide if a class is going to act concurrently or not. In Eiffel, this is solved much more elegantly by letting the programmer attach the **separate** keyword to every single object.

5. *SCOOP model*

   - I did not try all SCOOP features, especially not those being discussed late in the lecture.
   - I'm more experienced with .NET multithreading and know little about signals and events one has to introduce for concurrency problems. This seems way easier with SCOOP.
   - The integration with O-O seems way more natural than introducing threads and wait handles.
   - Experts in concurrent programming might like though a deeper control when dealing with the concurrency primitives directly.

   Tools

   - I missed, or maybe I did not explore enough, an overview of what is implemented in scoop2scoopli and what is still missing.
   - The current implementation with code generation makes it more difficult to [understand] what is going on.
   - After all it is very impressive how little one has to care about concurrency; it just works (after some struggling)!
   - Some sort of one-step compile would be convenient, a simple integration of EiffelStudio and the command line.

6. *SCOOP model*

   - The initial learning curve for SCOOP is comparatively steeper.
   - The idea of using features as enclosures of critical sections is a bit disturbing and makes the code somewhat unreadable. Although academically sound in principle, it is convenient to see something like: $lock(santa)\ \{Santa.doSomething\}$
   - Use of separateness in the software should start from design and should not just be an implementation option. A design that is done with threads in mind would not necessarily be the optimal template to implement SCOOP. Say for example, the common Model–View–Controller architectural pattern. Here it is taken for granted that the View can start off multiple threads of task on the Controller. Sometimes in the object design, we may need to have multiple threads of action. Say a Database object with a Query method. It is usual that more than one client, tries to execute concurrent queries. If this object is made separate, it results in sequential execution.
   - It would be good to have a nice debugger and profiler for SCOOP. This is presumably easier to achieve as compared to debugger for multi-threaded code. Debugging should be based on the original code and not the pre-compiled one.

- Finally, to be more widely usable SCOOP should progress towards other programming languages like Java or C♯.

Tools

- The performance of scoop2scoopli seems impressive, considering what it does in the background.
- It is a pity that it is not integrated in the Eiffel studio compiler invocation.
- In real-life, scoop2scoopli should remain hidden from the user (not just its invocation, but also the programmer should not be required to deal with its output).

7. *SCOOP model*

- I like the simple Action-Object-Processor model of SCOOP. But it took me some work to finish the two tasks including proper cleanup.
- I had big problems to comprehend the consequences that a processor can't unlock itself (while it is executing one action). I needed some hours of debugging to find out the reason, why two active objects can't communicate with each other without an intermediate object (*PROCESS_CONTROLLER* in my solution).
- I don't like that separate objects must be passed as arguments to lock them. This leads to countless enclosing routines and additional indirections just to lock some objects. This forced me to design my primitives in three layers.
- An approach that forces me to use three layers to model one idea is quite a low-level approach. It reminds me of the early days when we had just pointers and no references and no smart pointers or garbage collectors (depending on the language).

Tools

- A concurrency library based on SCOOP should include primitives like the ones used here and find a way to generate and/or hide the boiler-plate code needed. The universal enclosing agent is surely a step in the right direction.

8. *SCOOP model*

- SCOOP is a high-level abstract concept for concurrency, it is well defined with several rules, and it needs only one keyword **separate** to make the whole thing work. Often, the more compact the rule, the more variants we can achieve. I think the three properties – mutual exclusion, condition synchronization and wait by necessity – make SCOOP a powerful tool for most concurrent tasks. Compared with Java multithreading — which requires low-level synchronization code for monitors like *while* (...) *wait*(); ... *notify* (); — SCOOP integrates the concept of Design by Contracts and condition synchronization into one, preconditions become wait conditions, this is done in a very natural and seamless way. On the other hand, like other mechanisms, SCOOP also suffers from inheritance anomaly problems.
- The main problem is to get used to the concept of SCOOP and to apply the "SCOOP way" of thinking. SCOOP is rather a new concept and to master it surely requires more exercises.

Additional mechanisms or abstractions that SCOOP should offer

- An additional mechanisms that would have been nice to have for this project is the possibility to suspend an object for at least a certain amount of time. The feature suspend should take as an argument an *INTEGER* that specifies the minimum wait time in milliseconds for example. This feature would be useful for objects that are waiting for some event to arrive but do not want to block until the event arrives because they want to do another activity (like sleeping in the case of Santa) if there is no event ready. This other activity may not take very long so the object doesn't want to check for an arriving event immediately. Instead it wants to wait for the rest of the polling time.

Tools

- Scoop2scoopli is a nice pre-processor tool. It would be better if it could be integrated into the compiler so that the user does not need to compile the project twice.

9. *SCOOP model*

   - We have mixed feelings. On the one hand, it's a fairly nice and clean concept fitting into the object oriented paradigm and thus allows using concurrency very naturally. On the other hand, simplicity often comes with the cost of flexibility. And this is what I think happens with SCOOP. For example mutual exclusion can only be done for the whole method and the programmer has no way to bypass this restriction. Having a simple standard behaviour is a good thing of course and therefore it might be a big plus to give the experienced and advanced developers the tools to be more flexible — even if the price to pay is as high as an additional keyword ;)

   Additional mechanisms and abstractions that SCOOP should offer

   - With respect to the problems we had because we often forgot to properly wrap the asynchronous calls, it might be a nice thing to do the wrapping by default. It should also support agents and tuples. This would have us allowed to come up with a cleaner and more type safe solution for the active object implementation.

   Tools

   - It seems to be a really nice tool. For example, the error messages were pretty useful, event though a warning in case of improperly wrapped calls on separate objects would have saved us a few hours of debugging. Of course it will be even nicer once it's directly integrated in the compiler.

12.4 Discussion

The course turned out to be an invaluable source of suggestions coming from people who are "fresh" to the approach presented in this dissertation. Several problems pointed out by the students have already been solved; comments concerning the tools have also been taken into account.

- The agent mechanism now supports fully asynchronous calls. This permits emulation of rendezvous synchronisation.
- Single separate calls do not require individual wrapping routines; a universal wrapper is provided.
- The type-checking mechanism ensures correct wrapping of separate calls.
- The integration of scoop2scoopli with EiffelStudio [105] should simplify the development of SCOOP applications.

In the future iterations of the course, we are planning to avoid overlapping with topics taught in other concurrency courses, and focus solely on SCOOP. This will permit covering a larger number of topics, including the practical use of the advanced mechanisms recently introduced in the model: fully asynchronous calls, parallel wait, resource pools, event-driven programming (see section 10.2).

We would like to thank Patrick Eugster and Volkan Arslan for their contributions. Patrick prepared the teaching material — lecture slides and exercise sheets — for the non-SCOOP topics and taught that part in 2005; some of his material was reused in 2006. Volkan gave a lecture on event-driven and real-time programming.

13

Critique and conclusions

STARTING from the initial SCOOP_97 design, we have developed a practical framework for object-oriented concurrency, based on the principles of Design by Contract. A number of steps have been necessary to achieve this goal:

- Our study of the relationship between object-orientation and concurrency, with a particular focus on the role of assertions, has shown that Design by Contract is beneficial for concurrent systems in that it supports specifying all the required conditions — including synchronisation — for a correct interaction between clients and suppliers. Taking the viewpoint that programs are concurrent in general, and sequentiality is just a special case of concurrency, we have proposed a generalised semantics of object-oriented mechanisms and contracts, applicable in both concurrent and sequential contexts. We have argued that the traditional sequential semantics is simply a specialised version of the generalised semantics.

- We have enriched Eiffel's type system to capture concurrency-related properties of objects and entities: their locality and detachability. Carefully designed type rules (see chapter 6) eliminate potential atomicity violations without restricting the expressiveness of the model.

- The semantics of feature call and feature application mechanisms have been refined to support selective locking and lock passing. Both refinements have increased the expressiveness of the model; lock passing turns out to be necessary for sound reasoning about separate feature calls.

- SCOOP has been brought to a full compatibility with advanced object-oriented mechanisms: genericity, polymorphism, dynamic binding, agents, and once features. We have discussed the impact of these mechanisms on SCOOP and vice-versa, and proposed refined type and semantic rules to accommodate them. The agent mechanism is particularly beneficial to SCOOP because it permits fully asynchronous calls (see section 9.3.4) which were not possible in the original model. Furthermore, agents enable the implementation of other useful mechanisms, e.g. generic enclosing routines (see section 9.3.4), resource pools, parallel waiting for multiple results, and rendezvous-style synchronisation (see section 10.2).

- We have explored the feasibility of modular reasoning about concurrent software using the proposed contract semantics. The Hoare-style rule 8.2.2, which unifies the treatment of synchronous and asynchronous feature calls and enables modular and sequential-like reasoning, has been introduced. The new rule is stronger than the original rule of

SCOOP_97 in that it permits reasoning about preconditions and postconditions involving separate calls on *controlled* entities (see section 8.2).

- We have implemented the model as an Eiffel library $SCOOPLI$ (see section 11.2). The *scoop2scoopli* compiler, which type-checks SCOOP code and translates it into pure Eiffel, has also been implemented. The supporting *CONCURRENCY* library provides such advanced facilities as: fully asynchronous calls, asynchronous handling of events, and parallel wait.

- The SCOOP approach has been taught in two iterations of a graduate course on concurrent programming at the ETH Zurich. We have devised teaching material for the course: lecture slides, exercises, and examples. The $SCOOPLI$ library and the supporting tools have been used in the course.

The theoretical and practical results of our research help simplify the construction of concurrent systems by bringing the O-O programming method for such software to a higher level of abstraction and convenience, and making it easy to understand, learn, and apply. We view the new semantics of contracts, the enriched type system, the integration with advanced O-O mechanisms, and the support for agents and full asynchrony as the most important outcome of this work.

SCOOP is a promising approach to building high-quality concurrent systems. Unlike most existing COOLs, it is a full-blown language supporting multiple inheritance, polymorphism, dynamic binding, and genericity. A number of issues, however, remain unsolved. This chapter takes a critical look at the proposed framework, discusses its applicability to languages other than Eiffel, points out its limitations, and gives an overview of future research topics.

13.1 Applicability to other languages

SCOOP has a strong Eiffel flavour. Right at the beginning of this dissertation, however, we argued that the results of our work could be directly applied to other languages that support Design by Contract, in particular JML/Java [80] and Spec♯ [20]. This section takes a look at this issue from the point of view of a language designer who wants to integrate the proposed technique with one of these languages. Several language features and their compatibility with SCOOP are discussed; we point out the necessary restrictions or modifications of the target languages.

JML/Java and Spec♯ are good candidates for such an exercise because they satisfy three essential requirements:

- *Object-oriented programming model*
  Both languages are (almost) purely object-oriented. Computation is based on feature calls; polymorphism and dynamic binding are supported; the latest versions of the underlying languages (Java for JML, C♯ for Spec♯) also include generics.

- *Static typing*
  Strong typing ensures type safety, i.e. the absence of certain kinds of run-time errors, in a sequential context. This is essential for providing additional safety and fairness guarantees when introducing concurrency.

- *Support for Design by Contract*
  Both languages permit the use of routine preconditions and postconditions, as well as class invariants. The syntax and the semantics of annotations differ somewhat from those of Eiffel but the support for DbC is sufficient to introduce SCOOP.

We also considered SPARK — a subset of Ada equipped with contracts and annotations for data flow and information flow analysis [18] — as a potential host language; however, the limited support for O-O techniques, e.g. the absence of genericity and dynamic binding, makes it less adequate for the purpose. Nevertheless, SPARK remains an interesting target (and a possible topic of further research) because of its practical applications in mission-critical software.

Type annotations

Additional type annotations required by SCOOP are straightforward to introduce in both target languages. The distinction between attached and detachable types (called *non-nullable* and *nullable* respectively) is already supported by Spec♯ and JML. By default, Spec♯'s types are nullable unless decorated with '!'; a compiler switch, however, changes this policy so that all types all non-nullable unless decorated with '?', in line with the SCOOP convention (except that '?' appears after the class type in a declaration). In JML, any entity (other than a local variable) of a reference type is implicitly declared as non-nullable; it can be explicitly made nullable by annotating it with the *nullable* modifier. A declaration is implicitly declared nullable when the (outermost) class or interface containing the declaration is adorned by the class-level modifier *nullable_by_default* ; implicit nullability may be overridden by decorating a given entity with the modifier *non_null*.

Processor tags require an extension of Spec♯'s and JML's syntax. In Spec♯, this could be avoided by using *attributes* (not to be confused with Eiffel's attributes which are referred to as *fields* in Spec♯) to decorate methods and arguments. However, attributes can be applied to fields, method arguments and method results, but not to other constructs of interest, e.g. type casts. Therefore, an extension of the syntax is necessary to provide a full support for SCOOP. A point of style: since angle brackets are used in JML and C♯ for specifying actual generic parameters, SCOOP annotations for processor tags may be a bit confusing in the presence of generically derived types, e.g.

separate <*x*.**handler**> *List*<*string*>

To avoid the confusion and increase code readability, square brackets may be used for processor tags, i.e.

separate [*x*.**handler**] *List*<*string*>

This problem is less acute in JML because additional type annotations are wrapped in special comments, thus syntactically distinct from Java code, e.g.

/*@ **separate** <*x*.**handler**> @*/ *List*<*string*>

Figure 13.1 shows an example routine with SCOOP annotations expressed in Eiffel, Spec♯ (with the convention that types not decorated with '?' are non-nullable), and JML.

```
-- Eiffel
my_buffer: separate <px> BOUNDED_QUEUE [INTEGER]

store ( a_buffer : separate BOUNDED_QUEUE [INTEGER]; i: INTEGER)
        -- Store i in a_buffer .
    require
        not a_buffer . is_full
    do
        a_buffer .put (i)
    ensure
        a_buffer .count = old a_buffer .count + 1
    end

// Spec#
public separate [px] BoundedQueue<int> myBuffer;

public store (separate BoundedQueue<int> aBuffer, int i)
    requires !aBuffer . isFull ;
    ensures aBuffer .count == old(aBuffer .count)+1;
{
    aBuffer .put(i);
}


// JML
public /*@ separate <px> @*/ BoundedQueue<int> myBuffer;

/*@ requires !aBuffer . isFull ;
  @ ensures aBuffer .count == \old(aBuffer .count)+1;
  @*/
public store (/*@ final separate @*/ BoundedQueue<int> aBuffer, int i) {
    aBuffer .put(i);
}
```

Figure 13.1: SCOOP annotations in Spec♯ and JML/Java

Static features

The refined semantics of feature call and feature application (definitions 6.1.3 and 6.1.4 respectively) can be applied directly, with the exception of *static* methods and fields supported by both JML and Spec♯ (the latter also supports static *properties*) but prohibited in Eiffel. Static features belong to the declaring class rather than an instance; as a result, they are shared by all the instances of the given class. A class object handles all the static features of a given class; using a static feature requires the access to the corresponding class object. This raises a problem in a concurrent context: if client objects handled by different processors are calling static methods or assigning to static fields of the same class simultaneously, the mutual exclusion

guarantees are broken. To avoid this, one could simply prohibit static calls; however, two less radical alternatives exist:

- *System-wide class objects; implicit locking.*
  Use a dedicated processor for handling the class object. Mutual exclusion is enforced through implicit locking of that processor on each access to a static field or call to a static method. This solution is simple and preserves the original syntax of static calls but, besides introducing some run-time overhead, the implicit locking increases the danger of deadlock and is difficult to control and analyse.

- *Processor-wide class objects; no locking.*
  The same class object is shared by all instances of a given class handled by the same processor. Since the class object is non-separate from these instances, no additional locking is necessary to ensure mutual exclusion. The solution preserves the original syntax of static calls but static fields lose their singleton status. The amount of used memory space may be higher than in the previous solution: for each class, there may be as many class objects as processors. The absence of implicit locking, however, means that there is no run-time overhead.

The latter solution seems more suitable because it avoids implicit locking. It also has a familiar flavour: *once functions* of non-separate types are handled in a similar manner (see section 9.4).

Assignment to formal arguments

The synchronisation mechanism of SCOOP relies on the fact that formal arguments are non-writable. This is true in Eiffel; C♯ and Java, however, allow for assignments to formal arguments. The following example in JML, if accepted, may introduce traitors.

```
// JML
public badStore(/* @ separate @*/ BoundedQueue<int> aBuffer, int i) {
    aBuffer = new /*@ separate @*/ BoundedQueue<int>;
    aBuffer.put(i);        // traitor
}
```

The assignment to *aBuffer* turns it into a traitor: the formal argument now denotes a fresh buffer object located on a different processor than the original argument's handler. The subsequent call *aBuffer*.*put*(*i*), although valid according to the call validity rule 6.5.3, targets an uncontrolled entity. To ensure the soundness of the call validity rule, formal arguments must be non-writable. In JML, this can be achieved by requiring every formal argument to be declared as **final**; in Spec♯ the type rules must exclude formal arguments as targets of assignments.

Strictly speaking, only formal arguments of reference types need to be considered. Arguments of value types are not a problem because they are always seen as non-separate (see the explanation below) thus cannot become traitors. Nevertheless, to preserve full compatibility with SCOOP as defined for Eiffel, we suggest that all formal arguments be non-writable.

Value types

Value types such as *int*, *bool*, *float*, *double*, *char*, and *string* are treated like expanded types, i.e. they are always passed by copy and viewed as non-separate in any context. Rule 6.10.1

should apply to value types (for example prohibiting the declaration of separate entities of a value type); type combinators '$\star$' and '$\otimes$' (see figure 6.14) need to be adapted accordingly to ensure the invariance of value types under combination.

Assignment to attributes

Direct assignment to attributes of another object, of the form $x.a = a$, is supported in JML and Spec♯. Such assignments complicate static reasoning about object state, in particular the preservation of invariants; therefore they are prohibited in Eiffel. Nevertheless, the more flexible techniques for proofs of invariants used in JML (e.g. ownership and visibility methods by Müller et al. [102]) and Spec♯ (the Boogie methodology Barnett et al. [19]) allow safe handling of direct assignments to attributes.

Spec♯'s *properties* mechanism permits the use of the familiar attribute assignment as shorthand for a call to an appropriate setter feature, e.g.

```
// Spec#
public r(separate X x) {
    x.a = a;      // it really means x.setA(a)
    x.b = b;      // it really means x.setB(b)
}
```

JML does not provide this facility.

Direct assignments to attributes can be easily supported in SCOOP by viewing them — for the purpose of type-checking — as calls to implicit setter features which take one formal argument of the target attribute's type. Two important conditions must be satisfied to use this construct safely in a SCOOP-enabled context.

- Assignments to an attribute of x are permitted only if x is controlled. This follows from the call validity rule 6.5.3.
- The type of the target expression must be evaluated using the combinator '$\otimes$' rather than '$\star$' because it denotes the type of the implicit setter's formal argument as seen by the client object, and not the type of the attribute. For example, if the attribute a in class X has the type $(?, \bullet, A)$, i.e. it is non-separate, then the source of the assignment $x.a = a$ must conform to $(?, x.\mathbf{handler}, A)$ because

  $$(!, x.\mathbf{handler}, X) \otimes (?, \bullet, A) = (?, x.\mathbf{handler}, A)$$

  A calculation using '$\star$' — thus yielding $(?, \top, A)$ — might lead to an unsound assignment, and introduce a traitor.

Casts

The semantics of *type casts* differs somewhat from that of an object test as defined in section 6.7. Their treatment of detachable and processor tags, however, is identical: casts should take into account the voidness of the source entity, and the identity of the processor handling the object represented by that entity. A cast should succeed only if the source object's handler conforms to the processor tag of the target type. Expressions of the form *e* **instanceof** *T* in

JML and *e* **is** *T* in Spec♯ yield true if the value of *e* is not void and the corresponding cast would succeed.

Contracts

The new semantics of contracts, described in chapter 8, can be readily used in JML and Spec♯. As a result, preconditions have the wait semantics whereas postconditions, loop assertions, and checks are evaluated asynchronously if possible; the distinction between *controlled* and *uncontrolled* clauses applies. No additional rules, beyond those introduced in this dissertation, are necessary.

One mechanism of JML, however, needs a closer inspection. JML permits the specification of multiple precondition-postcondition pairs for the same feature, as shown in figure 13.2. Feature *storeIfPositive* stores the integer argument *i* in the buffer if *i* is positive; the be-

```
/*@ requires i>0 && !(aBuffer.isFull);
  @ ensures aBuffer.count == \old(aBuffer.count) && aBuffer.item == i;
  @ also
  @ requires i<=0;
  @ ensures true;
  @*/
 storeIfPositive (/*@ final separate @*/ BoundedQueue<int> aBuffer, int i) {
     ...
}
```

Figure 13.2: Multiple precondition-postcondition pairs in JML

haviour of the feature is undefined otherwise. The first **requires** clause may be uncontrolled (depending on whether the actual argument corresponding to *aBuffer* is controlled or not); the second one is always controlled. If the routine is called in a context where *i* is positive, then the second clause *i*<=0 does not hold; the processor applying *storeIfPositive* will block waiting for the first clause to become true. Such multiple precondition-postcondition pairs open up an interesting possibility: routines may specify a range of requirement levels through different precondition clauses. When a routine is applied, its executing processor waits until one of the preconditions is satisfied. The strongest satisfiable clause is chosen; if it does not hold, the next strongest is checked, and so on. This introduces an additional level of routine adaptability: properties established by a routine can now be adapted to the current state of the accessed resource. From the point of view of the proof technique outlined in section 8.2, the obligations on a client of the routine are weakened: it has to satisfy the disjunction of controlled precondition clauses coming from different precondition-postcondition pairs. The guarantees given to the client, however, are also weaker: only the disjunction of the controlled postcondition clauses from different pairs can be assumed.

It should be noted that Spec♯ insists on the use of side-effect free expressions in contracts to avoid any influence of contract checking on the semantics of correct programs. Procedural abstraction is allowed as long as any function occurring in a contract is *pure*. A syntactic notion of purity is overly restrictive, so different notions of behavioural purity are being explored [21, 106]. In the context of SCOOP, function calls with uncontrolled arguments may cause

waiting thus violate the purity requirement; one could prohibit such calls in assertions occurring in Spec♯ programs.

Polymorphism

Redefined methods inherit the original preconditions and postconditions. JML applies the same policy as Eiffel: preconditions may be kept or weakened; postconditions may be kept or strengthened. Spec♯ allows postcondition strengthening but precondition weakening is not supported. (To avoid weakening preconditions, Spec♯ only allows multiple inheritance from interfaces where all the inherited preconditions for a method are the same.)

The redefinition policy for attributes and routine arguments in JML and Spec♯ simplifies the treatment of polymorphism and dynamic binding in a concurrent context. Redefined features and entities preserve their original types. Problems discussed in section 9.1 — invalid precursor calls and inherited preconditions — disappear if detachable and processor tags obey the invariant redefinition policy; rules for feature redefinition in both host languages need not be modified.

Genericity

Genericity has been recently introduced in Spec♯ and Java. The mechanism in supported in Spec♯ but it may undergo some modifications, in particular with respect to detachable and attached types [55]. The current subtyping rules for generically derived types in Java allow covariance on the type but require identical generic parameters, e.g. $D\langle T\rangle$ conforms to $C\langle T\rangle$ if D conforms to C; but $D\langle U\rangle$ only conforms to $C\langle T\rangle$ if D conforms to C, and U and T are identical. (SCOOP would also allow $U \preceq T$ if both U and T are detachable; see rule 9.2.3.) Applying the same policy to detachable and processor tags preserves type safety but restricts the flexibility of the mechanism. We are not aware of any work on genericity for JML that takes into account detachable and attached types. It seems, however, that refined conformance rules for generically derived types, in line with the rule 9.2.3 proposed in this dissertation but disallowing class type covariance of formal generic parameters, could relax the above restrictions while preserving soundness.

Agents

SCOOP owes much of its expressive power to the support for agents. Spec♯ provides a similar mechanism for representing features ready to be called: *delegates*. Delegates are less flexible than agents: they do not offer the possibility to choose whether an actual argument is provided at creation or call time; only the target of a delegate may be left open. But the mechanisms is powerful enough to support fully asynchronous calls (see section 9.3.4) and the derived mechanisms described in section 10.2. Recent work by Müller and Ruskiewicz [104] demonstrates that delegates can provide more static type safety than agents. (Recall that the SCOOP's agents are only safe up to the safety offered by the mechanism in sequential Eiffel; see section 9.3.)

JML does not offer any similar mechanism. Full asynchrony can still be achieved there by implementing manually the pattern used in the feature *asynch* in class *CONCURRENCY* (see figure A.1). Unfortunately, without an agent-like mechanism, it is impossible to implement this

and the derived mechanisms as reusable components.

Modular invariants

The classical specification and verification method for class invariants used in Eiffel supports verification of object invariants that depend only on the attributes of the current object and non-mutable state of other objects (e.g. constants and *once functions*). Since SCOOP's call validity rule 6.5.3 prohibits separate calls in invariants, the approach classical approach remains unchanged in the presence of concurrency.

But the classical approach has its limitations, e.g. it does not allow layering of objects where the invariant of an object depends on the mutable state of other objects. Spec♯ and JML provide more flexible techniques for the verification of invariants that support object layering. Müller et al. [102] propose two approaches:

- *Static ownership technique*
  The hierarchical structure enforced by a static ownership model, e.g. the Universe Types [103] already supported by JML, relaxes the classical restriction on invariants; an invariant can now depend on the mutable state of all the objects owned by the current object (including that object itself). This technique is a proper generalisation of the classical approach; in addition to all the cases handled by the classical approach, it permits expressing invariants of non-trivial layered object structures, e.g. lists, hash tables, trees, provided these structures are properly encapsulated in an ownership universe.

- *Visibility technique*
  This technique is more permissive in that it allows dependencies between objects located in the same universe, i.e. the invariant of object *o1* may depend on the mutable state of another object *o2* if both objects reside in the same universe (they are *peers* in the Universe Types terminology); *o1* does not have to own *o2*. Additionally, recursive dependencies between objects are permitted.

The *Boogie methodology* proposed by Barnett et al. [19] and used in Spec♯ [20] relies on dynamic ownership. An object's invariant may depend on the mutable state of all objects it owns (transitively). Modifications of the object's state are only possible after performing an **unpack** operation. The object's invariant does not need to hold after that, so that assignments to attributes are safe. On putting the object in the immutable state — by performing a **pack** operation — its invariant must hold, and all the objects it depends on must be in a consistent state, i.e. packed.

The introduction of SCOOP into JML and Spec♯ does not impose additional restrictions on invariants. On the contrary: the support for modular invariant specification enables relaxing these restrictions. Calls of the form $q\ (x)$, where x is separate, could be allowed provided that x is owned by the current object, because the ownership of x allows assuming that no other clients may change its state between two consecutive evaluations of the invariant. (Recall that the proof rule 8.2.2 assumes the absence of such call.) Implicit locking and waiting involved in the evaluation of such invariants may lead to deadlocks but the problem may be solved by imposing a stricter ownership hierarchy. This topic needs further study but the static ownership technique seems to be the most appropriate for that purpose.

Run-time overhead

The cost of SCOOP support in JML and Spec♯ is similar as in Eiffel, i.e. every object needs an additional field to store a reference to its processor. This is required both for synchronisation purposes and for run-time type casts. Objects of value types do not need that field because they are non-separate, i.e. handled by the current processor, in every context.

Discussion

It seems that SCOOP can be integrated with JML/Java and Spec♯ in a straightforward manner. The similarity of object models and contract mechanisms to Eiffel's simplifies the task; syntactic differences are a minor issue. Spec♯ already has all the necessary facilities to support the advanced SCOOP mechanisms; JML/Java only lacks agents. Few restrictions on the host languages are necessary: essentially, assignments to formal arguments must be prohibited, and the semantics of static fields must change so that they are processor-wide rather than system-wide singletons.

Beyond illustrating the generality of the concurrency model proposed in this dissertation, the integration of SCOOP with JML and Spec♯ may bring more benefits, particularly in terms of tool support and language operability.

Static verification

A number of tools have been developed for static verification of JML programs. ESC/Java and ESC/Java2 [47, 40] can detect certain common errors and check simple assertions. JACK [36] offers similar functionality to ESC/Java; it is used for proving the correctness of Java Applets. CHASE [39] automatically checks frame conditions. LOOP [141] translates code annotated with JML specifications to proof obligations which are then submitted to the theorem prover PVS. LOOP handles more complex specifications and code than the above automatic checkers.

Spec♯s static program verifier Boogie constructs a program in its own intermediate language, BoogiePL [45]. From the BoogiePL program, Boogie infers loop invariants and generates verification conditions that it passes to an automatic theorem prover. Counterexamples reported by the theorem prover are translated back into error messages about the source code.

At the moment, these tools are unable to handle asynchrony, although ESC/Java2 has some support for multithreading. The introduction of SCOOP would require their extension, so that separateness of types is taken into account, and controlled contract clauses be treated as usual contracts without separate calls, with uncontrolled ones being ignored. The complexity of such extensions is difficult to assess; one may expect, however, that it should be simpler than the treatment of multithreading [128, 69].

Language interoperability

Both Spec♯ and Eiffel target the .NET platform; this permits mixing these languages in the same application. The interoperability is not limited to using classes and features implemented in different language; Spec♯ classes should be able to inherit from Eiffel classes and vice-versa. This

does not pose any bigger problems in a sequential context. The interesting question, however, is how to make the SCOOP-based concurrent extensions of both languages work together.

The Elevator.NET application described in section 10.3.1 uses a mix of SCOOP/Eiffel and C♯ code; the latter implements the GUI. In that case, however, C♯ code is not aware of SCOOP; it does not perform use any separate entities or perform any separate calls. In a way, the effects of C♯ code are limited to the boundaries of a single processor.

The real challenge is to make SCOOP-enabled code written in Eiffel and Spec♯ work seamlessly together. Can the differences between the two languages be accommodated in a concurrent context?

13.2 Limitations and future work

Although SCOOP is a full-fledged O-O framework supporting several advanced mechanisms and techniques, it has a number of limitations. This section discusses topics that fall beyond the scope of this dissertation but are essential for a general approach to concurrency.

Operational semantics

We have discussed the properties of the type system for SCOOP (see section 6.12) but provided no formal justification of its soundness. A formal study of SCOOP, including the development of a full operational semantics and a proof of type soundness, would constitute a rich PhD topic in itself. The formalisation of SCOOP_97 in CSP, proposed by Brooke et al. [34], covers a small subset of the model; it is insufficient for that purpose. We are currently working on modelling SCOOP programs with fair transition systems [88]. Ostroff et al. [114] use such a model for reasoning about properties beyond contracts in SCOOP_97 programs; their approach may be extended to cover a large subset of SCOOP. For the moment it is unclear how advanced mechanisms, e.g. agents and genericity, can be supported in that model.

Deadlock-freedom

This dissertation focuses on safety properties; some liveness properties, e.g. loop termination, have also been considered. The relation between deadlocks and contract violations has been studied in chapter 8; techniques which reduce the potential for deadlocks have been introduced in chapter 7. A run-time mechanism for deadlock detection has been devised and implemented as an extension of the *SCOOPLI* library by Moser [100]. Nevertheless, our type system allows for deadlocks; no method for preventing deadlocks or proving their absence has been proposed. An earlier attempt at solving this issue is described in [108]; however, that technique has been discarded due to its complexity and lack of modularity.

An extension of the contract mechanism is necessary to permit a specification of resources that a given feature may use. This work should be kept in synch with the ongoing development of techniques for effect specifications, e.g. Dynamic Contract Frames by Schoeller and Ostroff [132]. The main challenge remains the modularity of the anti-deadlock technique.

Exception handling

We have not discussed the interplay between concurrency and the exception mechanism of Eiffel. Within the context of a single processor, exceptions may be handled like in sequential Eiffel, i.e. the **rescue** clause of a routine should clean up the state of the current object (by establishing its invariant) and retry the execution or propagate the exception up the call chain. A problem arises at the processor's boundary: if the exception is not handled before that level, where and how should it be propagated? If the separate client is still in the context of the routine that locked the target processor, it can receive the exception and handle it as usual. Due to the presence of asynchrony, however, a faulty client might have already left the context, so that the exception cannot be propagated. A number of solutions are possible:

- The object where the exception occurred is marked as "dirty"; any subsequent access to that object causes the propagation of the pending exception. Other objects located on the target processor remain available as usual.
- The whole processor is marked as "dirty"; any subsequent accesses to that processor fail, with the pending exception being propagated to the client which performs the access.
- The processor is only accessible to the guilty client object; all the other objects view the processor as busy. On next access to the processor, the guilty client gets the pending exception.
- The processor is only accessible to the objects located on the same processor as the guilty client; other processors view it as busy. The propagation of the pending exception is like in the previous solution.
- The pending exception is sent to a global exception handler which grabs the guilty client as soon as it becomes available, and propagates the exception.

All these approaches have some advantages and disadvantages. The solution where the target processor is viewed as busy by the non-guilty processors combines best with the rest of the SCOOP mechanism. An extension of Eiffel's exception mechanism recently proposed by Arslan and Meyer [11] goes in that direction. An earlier study by Nenning [107] suggested a *wait on rescue* semantics to reduce asynchronous exceptions to synchronous ones; that approach, however, entailed a massive reduction of parallelism hence was not retained.

Real-time programming

Real-time programming with SCOOP is a topic of another PhD project in our group [9]. Our recent article [10] describes the combination of SCOOP and event-driven programming for modelling real-time applications; that approach relies partly on the extension of class *EVENT_TYPE* described in this dissertation (see sections 10.2.4, 11.4.7, and figure A.10). A number of facilities seem necessary for real-time programming:

- *Timeouts*
  Clients should be able to specify the maximum waiting time for separate resources. Preconditions may include assertions of the form *timeout* (*t*) where *t* is the maximum waiting time expressed in milliseconds. A call to a routine including such a precondition

clause succeeds if, within the specified deadline, the arguments are available and the precondition holds; otherwise, the call fails. This mechanism may be extended so that *timeout* (0) means "available immediately". In a way, timeouts are guarantees for the client, so placing their specifications in preconditions may not be the best choice. All the ramifications of this mechanism should be analysed, in particular its soundness in the context of inheritance and polymorphism which allow precondition weakening. Alternative mechanisms for timeouts may be considered.

- *Worst-case execution time (WCET)*
  Clients should be able to assess the worst-case execution time of a feature call. The worst-case execution time of a call may be calculated from the maximum waiting time (see the above bullet point) and the execution time of the routine's body. The latter may be specified using a postcondition clause of the form *wcet* (*t*). Here too, the exact placement of specifications needs to be justified; postcondition-based specifications seem appropriate but it is not clear yet how well they combine with the rest of SCOOP. Another open question is the specification of worst-case execution times in the presence of recursion and separate callbacks.

- *Priorities*
  The fair scheduling policy applied in SCOOP and implemented by SCOOPLI is too restrictive: it does not permit overtaking a request by another one. In real-time systems, however, requests with a higher priority should be allowed to overtake those with a lower priority. To satisfy this requirement, a more flexible policy (or a set of such policies) has to be implemented. This means sacrificing the fairness of SCOOP but is necessary to provide practical support for real-time computation.

  Prioritisation at the level of CPU time sharing between processors is also important. The current implementation leaves the management of the CPU time to the operating system's scheduler and, besides fairness, makes no further assumptions about its algorithm. In a real-time context, the application should retain control over low-level scheduling; this usually requires the use of a custom-made scheduler. Andreas Compeer's Master thesis tackles this issue.

- *Preemption*
  The basic SCOOP model does not support preemption, i.e. a client cannot be forced to relinquish a resource it currently holds. The absence of such a facility would be dangerous in a real-time application because a single badly-behaved client could break the whole computation — making other clients miss their deadlines — by keeping a resource for too long. A new mechanism is needed to enable preemption. SCOOP_97's *duels* (see section 4.2.7 and [94]) provide this facility; but the mechanism is too limited and it does not combine well with other necessary techniques, in particular timeouts and priorities.

Many of the above mechanisms rely on an appropriate exception handling, discussed in the previous section.

Requirements for a real-time programming framework differ considerably from those identified for the general SCOOP model. For example, fair scheduling is usually not required, as long as all requests meet their deadlines. Similarly, modularity is not a paramount issue in real-time systems; such systems are usually composed of a fixed number of elements and one can

assume the knowledge about the whole program. By sacrificing modularity, several difficult issues can be solved: deadlocks are easier to tackle using a type system, reasoning about execution time also becomes easier. Nevertheless, the integration of additional mechanisms needed in a real-time context with advanced O-O techniques is a real challenge. SCOOP seems to provide a good basis for such efforts but the feasibility of real-time programming with SCOOP remains to be demonstrated.

Distributed programming

One of the goals of SCOOP is to bring distributed programming to a higher level of abstraction and convenience, for example by hiding the details of physical object distribution. Processor tags in SCOOP types identify *abstract* processors handling objects represented by entities. The physical location of an object's handler is transparent to the programmer; there is no difference between calls to processors located on a local machine, another machine in a local network, or a remote machine accessible via the Internet.

A number of mechanisms for distributed programming have been introduced in this dissertation. For example, explicit processor tags make it is possible to create objects on a specified processor without using a specialised remote factory object (see section 6.6). The operations *import*, *deep_import*, *flat_import* , and *independent_import* permit copying remote object structures and handling them as local ones (see section 6.8). Nevertheless, our implementation only targets single machines; the practical use of SCOOP in a distributed context remains an open issue. A successful feasibility study of a distributed implementation was done by Petrovay [121]; however, it was limited to subset of SCOOP_97 and excluded advanced mechanisms supported by SCOOP.

Distributed programming in SCOOP is the most important topic of further research. Distributed computation raises a number of challenging issues. Of particular interest are:

- *User-defined mapping of processors to physical resources*
  Transparent distribution of processors is convenient because it supports portability of SCOOP programs. In many situations, however, it is important to make full use of the available resources, e.g. multiple CPUs, clusters of machines, or local networks, by defining a precise mapping of abstract processors to the physical computation nodes. Custom mapping of resources is particularly important for heavy computations where load balancing is essential.

  SCOOP_97 proposed the Concurrency Control File (CCF) mechanism for that purpose (see section 4.2.7). CCF files are separated from the program code; an application is compiled without any reference to a specific hardware or network architecture. At run time each component of the application uses a CCF to find the available local and remote computing resources.

  The CCF mechanism permits load balancing but is limited in that programmers have no direct control over resources; therefore, we have discarded it in SCOOP. The physical distribution should be specified directly in the program code, e.g. using specialised library calls. An additional layer of abstraction in the form of a CCF-like file or a naming service advocated by Petrovay [121] may be used but it should only complement the code-based control.

- *Distributed scheduling*
  In our implementation, all locking requests are handled by a centralised scheduler which applies a fair policy: satisfiable requests are handled in the FIFO order (see section 11.2.4). In a distributed context, centralised scheduling may be very inefficient due to the communication latency: a client sends a request to the scheduler and waits; the scheduler sends a notification to the client when the request has been handled. In between, when checking the availability of resources, the scheduler performs additional messaging. All these message round-trips, possibly over slow long-distance links, introduce large overhead; particularly if the client processor and the requested processors run within the same local area network while the scheduler resides elsewhere. In that situation, a local scheduler handling accesses to processors within the local network would be much more efficient. The existence of many local schedulers within one SCOOP system, however, raises the question of consistency and fairness. How to handle requests for several processors located in different networks? How to prevent starvation of requests targeting a processor located in a different network?

- *SCOOP runtime support*
  Creating a new processor on a remote machine requires the existence of a SCOOP runtime that can take care of handling the processor, finding the nearest scheduler, etc. A SCOOP "daemon" could be required on target machines. It is not clear how SCOOP runtimes on different operating systems can work together

Shared access to resources

SCOOP enforces strict mutual exclusion: only one client processor may access the same target processor at any time. The lack of intra-object concurrency simplifies the model and formal reasoning about programs but certain synchronisation patterns, e.g. *readers-writers*, cannot be directly implemented in SCOOP.

The locking policy introduced in chapter 7 is derived from our earlier work on locking in SCOOP [110]. That work includes a basic mechanism for shared locking, based on a refined notion of a *pure query* and a new semantics for **only** clauses appearing in routine postconditions. Such clauses are used in sequential Eiffel to express the frame properties of features; we extend them to include processor tags, to permit support specification of purity with respect to a given processor. A query is pure with respect to a processor only if it does not modify the data structure handled by this processor. Simultaneous execution of pure queries on targets handled by the same processor is allowed; deep down, their application is serialised, so the sequential character of processors is preserved (there is no intra-object concurrency). Calls to features other than pure queries obey the standard mutual exclusion policy. *Shared locks* are necessary to support the proposed extension. A shared lock the application of one or more pure queries on the locked processor. Execution of non-pure features is prohibited when a shared lock is used. A client object can obtain a shared lock even if another client object is already holding a shared lock on the same resource. If a client object holds an exclusive lock on a resource, no other clients can obtain locks on that resource.

The extension proposed in [110] preserves the safety guarantees offered by SCOOP. But it does not support polymorphism; furthermore, the proof technique outlined in section 8.2 is not applicable. Therefore, we have excluded the mechanism from the framework described in

this dissertation. Nevertheless, we are planning to refine the shared locking technique so that it is fully compatible with other mechanisms supported by SCOOP. We expect the resulting approach to overlap with the future technique for deadlock prevention (see sections above).

13.3 Final remarks

In no way is the SCOOP framework a panacea for all the problems of concurrent programming. But we did not aim at it; the goal was to explore the possibilities offered by the well-established O-O techniques and Design by Contract. A number of important issues have been solved, bringing concurrent programming to a higher level of abstraction and convenience, and challenging many misconceptions prevailing in the industrial and academic world:

- *Development of concurrent software is difficult.*
  We have argued that the apparent difficulty is due to the insufficient adaptation and use of well-founded techniques, such as object technology and Design by Contract, and the lack of supporting tools.

- *Concurrency and O-O techniques do not combine well.*
  SCOOP indicates that there is no clash between concurrency and object technology; in fact, the full power of object technology can only be unleashed in a concurrent context. Both concurrency and object orientation are necessary components of a practical software development method.

- *Concurrent systems cannot be built and analysed in a modular fashion.*
  Our approach makes it possible to construct software elements which can be used as building bricks for larger systems. DbC enables specifying clear interfaces of software components; all interaction between the components is based on these interfaces. Abstraction and encapsulation shield each component from harmful interference. They also facilitate modular reasoning about software.

- *Existing software, in particular "sequential" libraries, cannot be easily reused in a concurrent context.*
  We have shown that existing library classes can be used as building blocks for many concurrent systems, independently of whether they have been written for such use. The contract-based synchronisation policy of SCOOP reduces the programming burden by providing the necessary safety and fairness guarantees; programmers can focus on writing algorithms without guessing all the possible contexts (number of clients, sequential or concurrent accesses, etc.) in which a given class will be used. As a result, the amount of reuse achieved in concurrent systems may match that of sequential ones.

It is believed that a PhD thesis should spawn more interesting research problems than it solves. I hope that my dissertation satisfies this requirement and constitutes a solid basis for further efforts aiming at a deeper understanding of object-oriented concurrency, to foster abstraction, simplicity, and modularity.

My research journey ends here, although many more places are to be visited yet. During this work I have discovered the beauty of object technology, its strengths and limitations, and the richness of its applications. But most importantly, I have learnt one essential thing that will guide my future research efforts: **S**implicity **C**an **O**ften **O**bviate **P**roblems.

A

CONCURRENCY library

```
deferred class CONCURRENCY
feature -- Basic operations
    call (a_feature : separate ROUTINE [ANY, TUPLE])
            -- Universal enclosing routine
        do
            a_feature.call ([])
        end

    asynch (a_feature : ?separate ROUTINE [ANY, TUPLE])
            -- Execute a_feature fully asynchronously.
        require
            a_feature_exists : a_feature /= Void
        local
            executor: separate EXECUTOR
        do
            create executor.execute (a_feature)
        end

    sleep (a_time: INTEGER)
            -- Suspend activity for a_time milliseconds.
        require
            a_time_non_negative : a_time >= 0
        do
            -- Implementation provided by scoop2scoopli
        end

feature -- Waiting faster
    evaluated_in_parallel (a_queries: LIST [?separate FUNCTION
                                [ANY, TUPLE, ?separate ANY]];
                            an_initial_answer, a_ready_answer: ?separate ANY;
                            an_operator: FUNCTION
                                [ANY, TUPLE, ?separate ANY]): ?separate ANY
            -- Parallel evaluation of queries combined by an_operator
        require
            a_queries.count > 0
        local
            answer_collector: separate ANSWER_COLLECTOR
        do
            create answer_collector.make (a_queries, an_initial_answer,
                                          a_ready_answer, an_operator)
            Result := answer (answer_collector)
        end
...
```

Figure A.1: CONCURRENCY

```
...
    answer ( an_answer_collector : separate ANSWER_COLLECTOR): ?separate ANY
            -- Answer from an_answer_collector
        require
            an_answer_collector . is_ready
        do
            Result := an_answer_collector .answer
        end

feature -- Predefined parallel operators
    parallel_or (l: LIST [?separate PREDICATE [ANY, TUPLE]]): BOOLEAN
        do
            if {res: BOOLEAN} evaluated_in_parallel (l, False, True,
                                agent or_else (b1, b2: BOOLEAN): BOOLEAN
                                    do Result := b1 or else b2 end (?, ?))
            then Result := res end
        end

    parallel_and (l: LIST [?separate PREDICATE [ANY, TUPLE]]): BOOLEAN
        do
            if {res: BOOLEAN} evaluated_in_parallel (l, True, False,
                                agent and_then (b1, b2: BOOLEAN): BOOLEAN
                                    do Result := b1 and then b2 end (?, ?))
            then Result := res end
        end

    parallel_sum (l: LIST [?separate FUNCTION [ANY, TUPLE, INTEGER]]):
        INTEGER
        do
            if {res: INTEGER} evaluated_in_parallel (l, 0,
                                {INTEGER}.min_value,
                                agent sum (i, j: INTEGER): INTEGER
                                    do Result := i + j end (?, ?))
            then Result := res end
        end
...
```

Figure A.2: CONCURRENCY (continued)

```
...
feature -- Resource pooling
    call_m_out_of_n (a_feature: ROUTINE [ANY, TUPLE];
                     a_pool: LIST [?separate ANY]]; m: INTEGER)
            -- Apply a_feature to m elements of a_pool.
        require
            m > 0 and then a_pool.count >= m
        local
            pool_manager: separate POOL_MANAGER
            locker: separate LOCKER
        do
            create pool_manager.make (a_feature, m)
            from a_pool.start until a_pool.after loop
                create locker.try_to_lock (a_pool.item, pool_manager)
                a_pool.forth
            end
        end
end
```

Figure A.3: CONCURRENCY (continued)

```
class EXECUTOR inherit CONCURRENCY
create execute, apply_to_target
feature {NONE} -- Initialization
    execute (a_feature: ?separate ROUTINE [ANY, TUPLE])
            -- Execute a_feature.
        require
            a_feature /= Void
        do
            if {f: separate ROUTINE [ANY, TUPLE]}a_feature then
                call (f)
            end
        end

    apply_to_target (a_feature: separate ROUTINE [ANY, TUPLE];
                     a_target: ANY)
            -- Apply a_feature to a_target (synchronously).
        do
            a_feature.import.call ([a_target])
        end
end
```

Figure A.4: EXECUTOR

```
class ANSWER_COLLECTOR inherit CONCURRENCY
create make
feature {NONE} -- Initialization
    make (a_queries : separate LIST [?separate FUNCTION
                                     [ANY, TUPLE, ?separate ANY]];
          an_initial_answer , a_ready_answer: ?separate ANY;
          a_combinator: separate FUNCTION [ANY, TUPLE, ?separate ANY])
            -- Creation procedure.
        require
            a_queries.count > 0
        local
            evaluator: separate EVALUATOR
        do
            answer := an_initial_answer ; ready_answer := a_ready_answer
            count := a_queries.count
            combinator := a_combinator.deep_import -- Non-separate copy
            from a_queries.start until a_queries.after loop
                create evaluator.make (a_queries.item, Current)
                launch_executor (evaluator)
                a_queries.forth
            end
        ensure
            answer = an_initial_answer
            ready_answer = a_ready_answer
            count = a_queries.count
        end

feature {CONCURRENCY} -- Answer retrieval
    answer: ?separate ANY
        -- Answer

    is_ready : BOOLEAN
        -- Is answer ready?
...
```

Figure A.5: ANSWER_COLLECTOR

```
...
feature {EVALUATOR} -- Answer update
    update_answer ( a_result : ?separate ANY)
            -- Update answer with a_result .
        do
            count := count - 1
            if not is_ready then
                combinator.call ([answer, a_result ])
                answer := combinator. last_result
            end
            if count = 0 or else equal(answer, ready_answer) then
                is_ready := True
            end
        ensure
            count = 0 implies is_ready
        end

feature {NONE} -- Implementation
    combinator: FUNCTION [ANY, TUPLE, ?separate ANY]
        -- Combinator for results

    ready_answer: ?separate ANY
        -- Answer that allows ignoring further results

    count: INTEGER
        -- Number of partial results to come
end
```

Figure A.6: ANSWER_COLLECTOR (continued)

```
class EVALUATOR inherit EXECUTOR
create evaluate
feature {NONE} -- Initialization
    evaluate (a_query: ?separate FUNCTION [ANY, TUPLE, ANY];
              an_answer_collector : ?separate ANSWER_COLLECTOR)
            -- Creation procedure.
        require
            a_query /= Void
            an_answer_collector /= Void
        do
            if {a: separate ANSWER_COLLECTOR}an_answer_collector then
                answer_collector := a
            end
            if {q: separate FUNCTION [ANY, TUPLE, ?separate ANY}a_query then
                evaluate_and_report (q)
            end
        end

feature {NONE} -- Implementation
    evaluate_and_report (a_query: separate FUNCTION
                                  [ANY, TUPLE, ?separate ANY])
            -- Evaluate a_query and report result to answer_collector .
        do
            a_query.call ([])
            notify_answer_collector ( answer_collector , a_query. last_result )
        end

    notify_answer_collector
                ( an_answer_collector : separate ANSWER_COLLECTOR;
                  a_result : ?separate ANY)
            -- Report a_result to an_answer_collector .
        do
            an_answer_collector .update_answer ( a_result )
        end

    answer_collector : separate ANSWER_COLLECTOR
        -- Answer collector
end
```

Figure A.7: EVALUATOR

```
class POOL_MANAGER inherit CONCURRENCY
create make
feature {NONE} -- Initialization
    make (a_feature : separate ROUTINE [ANY, TUPLE]; i: INTEGER)
            -- Creation procedure.
        require
            i > 0
        do
            feature_to_apply := a_feature.import -- Non-separate copy
            m := i
        ensure
            m = i
            count = 0
        end

feature {LOCKER} -- Feature application
    try_to_apply_feature (a_target : separate ANY)
            -- Apply feature_to_apply to a_target.
            -- Do nothing if already applied the required number of times.
        local
            envoy: separate <a_target.handler> EXECUTOR
                -- Non-separate from a_target. Needed because open-target
                -- agents cannot be applied to separate targets.
        do
            if count < m then
                create envoy.apply_to_target (feature_to_apply, a_target)
                count := count + 1
            end
        end

feature {NONE} -- Implementation
    feature_to_apply : ROUTINE [ANY, TUPLE]
            -- Feature to apply

    m: INTEGER
            -- Requested number of executions

    count: INTEGER
            -- Number of executions already performed

invariant
    m > 0
    count >= 0 and then count <= m
end
```

Figure A.8: POOL_MANAGER

```
class LOCKER inherit EXECUTOR
create try_to_lock
feature {NONE} -- Initialization
    lock_target (a_target : ?separate ANY;
                 a_pool_manager: ?separate POOL_MANAGER)
            -- Creation procedure.
        require
            a_target /= Void
            a_pool_manager /= Void
        do
            if {p: separate POOL_MANAGER}a_pool_manager then
                pool_manager := p
            end
            if {t: separate ANY}a_target then
                lock_target_and_report (t)
            end
        end

feature {NONE} -- Implementation
    lock_target_and_report (a_target : separate ANY)
            -- Lock a_target and report to pool_manager.
        do
            report (pool_manager, a_target)
        end

    report (a_pool_manager: separate POOL_MANAGER;
            a_target : separate ANY)
            -- Ask a_pool_manager to apply requested feature to a_target .
        do
            a_pool_manager. try_to_apply_feature (a_target)
            -- Lock passing occurs here.
        end

    pool_manager: separate POOL_MANAGER
        -- Resource pool manager
end
```

Figure A.9: LOCKER

```
-- Notion of event type; extended to provide SCOOP support
-- Original class by Volkan Arslan; SCOOP extensions by Piotr Nienaltowski

class EVENT_TYPE [?EVENT_DATA -> separate TUPLE create default_create end]
inherit
    LINKED_LIST [?separate ROUTINE [ANY, EVENT_DATA]]
        redefine default_create end
    CONCURRENCY

feature {NONE} -- Initialization
    default_create
        do
            make
            compare_objects
        end

feature -- Element change
    subscribe (an_action: ?separate ROUTINE [ANY, EVENT_DATA])
            -- Add an_action to subscription list.
        require
            an_action_not_void: an_action /= Void
            an_action_not_yet_subscribed: not has (an_action)
        do
            extend (an_action)
        ensure
            an_action_subscribed: count = old count + 1 and has (an_action)
            index_at_same_position: index = old index
        end

    unsubscribe (an_action: ?separate ROUTINE [ANY, EVENT_DATA])
            -- Remove an_action from subscription list.
        require
            an_action_not_void: an_action /= Void
            an_action_subscribed: has (an_action)
        local
            pos: INTEGER
        do
            pos := index
            start
            search (an_action)
            remove
            go_i_th (pos)
        ensure
            an_action_unsubscribed: count = old count-1 and not has (an_action)
            index_has_not_moved: index = old index
        end
...
```

Figure A.10: SCOOP-enabled EVENT_TYPE

```
...
feature -- Event publication
    publish (arguments: EVENT_DATA)
            -- Notify all actions from subscription list .
        require
            arguments_not_void: arguments /= Void
        do
            if not is_suspended then
                from start until after loop
                    if {action: separate ROUTINE [ANY, EVENT_DATA]}item then
                        asynch (agent action.call (arguments))
                        -- Full asynchrony; no waiting here
                    end
                    forth
                end
            end
        end

feature -- Status report
    is_suspended: BOOLEAN
            -- Is publication of all actions suspended? (Default: no.)

feature -- Status change
    suspend_subscription
            -- Ignore actions in subscription list
            -- until restore_subscription is called.
        do
            is_suspended := True
        ensure
            subscription_suspended: is_suspended
        end

    restore_subscription
            -- Consider again actions from subscription list
            -- until suspend_subscription is called.
        do
            is_suspended := False
        ensure
            subscription_not_suspended: not is_suspended
        end
end
```

Figure A.11: SCOOP-enabled EVENT_TYPE (continued)

B

Glossary

Actual argument
Expression *e* passed to a feature call *f* (..., *e* ,...) or *x*.*f* (..., *e* ,...) .
Java counterpart: actual parameter

Attribute
A *query* whose result is stored in memory.
Java counterpart: field

Client
A class that uses *features* of another class (its *supplier*).

Command
A *feature* that may change the state of the *target* but does not return a result.
Commands are implemented as *procedures*.

Contract
A set of conditions which govern relations between *clients* and *suppliers*.

Creation procedure
A procedure that initialises an object, used for object creation.
Java counterpart: constructor

Current
The *target* of *unqualified calls*.
Java counterpart: ***this***

Entity
An attribute, a local variable, or a *formal argument*.

Feature
An operation applicable to instances of a given class.

Formal argument
Entity *e* in a routine *r* (...; *e*: *T* ;...) .
Java counterpart: formal parameter

Function
A *routine* that returns a result.

Non-separate call
A feature call whose target is a *non-separate object*.

Non-separate object
An object handled by the same *processor* as the *current* object.

Procedure
A *routine* that does not return a result.

Processor
An autonomous thread of control capable of supporting sequential execution of instructions on one or more objects.

Qualified call
A feature call of the form $x.f$ or $x.f\ (...)$.

Query
A *feature* that returns a result but should not modify the state of the *target*.
Queries are implemented as *attributes* or *functions*.

Root class
A class whose instance is used as *root object*.

Root creation procedure
A *creation procedure* applied to the *root object* when a software system is executed.
Java counterpart: main method

Root object
The first object created by a software system; an instance of the *root class*.

Routine
A *feature* that performs some computation.
Java counterpart: method

Separate call
A feature call whose target is a *separate object*.

Separate object
An object handled by a different *processor* than the *current* object.

Supplier
A class whose *features* are used by other classes (its *clients*).

Target
The object on which a *feature* is applied.
Java counterpart: receiver object

Unqualified call
A feature call of the form f or f (...) . It targets the *current* object.

Bibliography

[1] Erika Abraham, Frank S. de Boer, Martin Steffen, and Willem-Paul de Roever. A Hoare logic for monitors in Java. Technical Report TR-ST-03-1, Christian-Albrechts-University Kiel, April 2003.

[2] Cyril Adrian. SCOOP for SmallEiffel. draft, available online at http://www.chez.com/cadrian/eiffel/scoop.html, June 2002.

[3] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[4] Gul Agha, Peter Wegner, and Akinori Yonezawa, editors. *Research directions in concurrent object-oriented programming*. MIT Press, Cambridge, Massachussetts, USA, 1993.

[5] Pierre America. Pool-t: A parallel object-oriented language. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[6] Pierre America. Designing an object-oriented language with behavioural subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, number 498 in LNCS. J. W. de Bakker, W. P. de Roever, and G. Rozenberg, May-June 1990.

[7] Pierre America. A parallel object-oriented language with inheritance and subtyping. *SIGPLAN Notices*, 25(10):161–168, October 1990.

[8] Gregory R. Andrews. *Foundations of multithreaded, parallel, and distributed programming*. Addison-Wesley, 2000.

[9] Volkan Arslan. Application of SCOOP to real-time systems. PhD proposal, August 2005. available at http://se.inf.ethz.ch/people/arslan.

[10] Volkan Arslan, Patrick Eugster, and Piotr Nienaltowski. Modelling embedded real-time applications with objects and events. In *IEEE RTAS*, San Jose, USA, April 2006.

[11] Volkan Arslan and Bertrand Meyer. Asynchronous exceptions in concurrent object-oriented programming. In *International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, San Jose, USA, April 2006.

[12] Volkan Arslan, Piotr Nienaltowski, and Karine Arnout. Event library: an object-oriented library for event-driven design. In *Joint Modular Languages Conference (JMLC)*, Klagenfurt, Austria, September 2003.

[13] Cyrille Artho. *Combining static and dynamic analysis to find multi-threading faults beyond data races*. PhD thesis, ETH Zurich, 2005.

[14] Isabelle Attali, Denis Caromel, and Sidi Ould Ehmety. Formal properties of the Eiffel// model. In *Parallel and Distributed Objects*. Hermes Science Publications, 1999.

[15] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Minneapolis, USA, October 2000.

[16] Arnaud Bailly. Formal semantics and proof system for SCOOP. White paper, October 2004.

[17] R. Balter, J. Bernadat, D. Decouchant, A Duda, A. Freyssinet, S Krakowiak, M Meysembourg, P. Le Dot, H. Van Nguyen, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, and G Vandôme. Architecture and implementation of Guide. *Computing Systems*, 4(1):31–67, 1991.

[18] John Barnes. *High integrity software: the SPARK approach to safety and security*. Addison-Wesley, 2003.

[19] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2004.

[20] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec♯ programming system: an overview. In *CASSIS*, volume 3362 of *LNCS*. Springer Verlag, 2004.

[21] Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: useful abstractions in specifications. In *6th workshop on Formal Techniques for Java-like Programs (FTfJP)*, June 2004.

[22] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[23] Mordechai Ben-Ari. How to solve the Santa Claus problem. *Concurrency: Practice and Experience*, 10(6):485–496, 1998.

[24] Erwin Betschart. Reimplementation of the elevator control application using EiffelVision. Semester thesis, ETH Zurich, Departement Informatik, 2005.

[25] Eric Bezault et al. The Gobo Eiffel library. http://www.gobosoft.com.

[26] F. Bodin, P. Beckman, D.Cannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of the 1993 Supercomputing Conference*, 1993.

[27] Emmanuel Bouyer and Gordon James. Eiflex: why we didn't use SCOOP. In *International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, pages 50–55, York, U.K., July 2006.

[28] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, USA, November 2002.

[29] Chandrasekhar Boyapati and Martin Rinard. A parametrized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[30] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[31] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent OOP. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 276 of *LNCS*, pages 33–40, 1987.

[32] Phillip J. Brooke and Richard F. Paige. An alternative model of concurrency for Eiffel. In *CORDIE*, pages 141–161, July 2006.

[33] Phillip J. Brooke and Richard F. Paige. A critique of SCOOP. In *CORDIE*, pages 56–61, July 2006.

[34] Phillip J. Brooke, Richard F. Paige, and Jeremy L. Jacob. A CSP model of Eiffel's SCOOP. submitted for publication, November 2005.

[35] Martin Büchi. *Safe Language Mechanisms for Modularization and Concurrency*. PhD thesis, Turku Centre for Computer Science, May 2000. available online at http://www.abo.fi/~mbuechi/publications/Thesis.html.

[36] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[37] Denis Caromel. Service, asynchrony, and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–18, 1989.

[38] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[39] Néstor Catano and Marieke Huisman. Chase: a static checker for JML's assignable clause. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 26–40, London, UK, 2003. Springer-Verlag.

[40] Patrice Chalin, Joe Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO)*, 2005.

[41] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[42] J. Choi, K. Lee, A. Loginov, R. OCallahan, V. Sarkar, and M. Sridharan. Efficient and precise data race detection for multithreaded object-oriented programs. In *Programming Languages Design and Implementation (PLDI)*, June 2002.

[43] Michael James Compton. SCOOP: An investigation of concurrency in Eiffel. MSc thesis, Department of Computer Science, The Australian National University, December 2000.

[44] P. Dasgupta, R. Ananthanarayanan, S. Menon, A. Mohindra, and M. Pearson. Language and operating system support for distributed programming in Clouds. In *Proceedings of the 2nd Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, March 1991.

[45] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[46] Alan L. Dennis. *.NET Multithreading*. Manning, Greenwich, USA, 2003.

[47] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[48] Werner Dietl, Peter Müller, and Sohia Drossopoulou. Generic Universe Types. In *FOOL/WOODS*, 2007.

[49] Werner Dietl, Peter Müller, and Arnd Poetzsch-Heffter. A type system for checking applet isolation in JavaCard. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, volume 3362 of *Lecture Notes in Computer Science*, pages 129–150. Springer-Verlag, 2004.

[50] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informaticae*, 1(2):115–138, June 1971.

[51] Edsger W. Dijkstra. Two starvation-free solutions of a general exclusion problem. EWD625, 1977.

[52] Sophia Drossopoulou and Susan Eisenbach. Describing the semantics of Java an proving its type soundness. In Jim Alves-Foss, editor, *Formal syntax and semantics of Java*, volume LNCS 1523, pages 41–80. Springer-Verlag, 1999.

[53] ECMA. Eiffel analysis, design, and programming language. ECMA Standard 367, June 2005.

[54] ECMA. C♯ language specification. ECMA Standard 334, fourth edition, June 2006.

[55] Manuel Fändrich. Non-nullable types and genericity in Spec♯. private communication, September 2006.

[56] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.

[57] Cormac Flanagan and Stephen Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[58] Cormac Flanagan and Shaz Quadeer. A type and effect system for atomicity. In *Programing Language Design and Implementation (PLDI)*, pages 338–349, 2003.

[59] Cormac Flanagan and Shaz Quadeer. Types for atomicity. In *SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 1–12, 2003.

[60] Nati Fuks, Jonathan S. Ostroff, and Richard Paige. SCOOP to Eiffel code generator. *Journal of Object Technology (JOT)*, 3(10):143–160, November–December 2004.

[61] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[62] L. Gunaseelan and R.J. LeBlanc. Distributed Eiffel: A language for programming multi-granular objects. In *4th International Conference on Computer Languages*, San Francisco, USA, 1992.

[63] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *ACM Conference on Principles and Practice of Parallel Programming (PPoPP)*, Chicago, USA, June 2005.

[64] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[65] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[66] C.A.R. Hoare. Concurrent programs wait faster. Technical Report, MSR Cambridge, July 2001. available online at http://research.microsoft.com/ thoare/.

[67] Matthias Humbert. Programming in SCOOP. Diploma thesis, ETH Zurich, Departement Informatik, 2004.

[68] ISO. Eiffel analysis, design, and programming language. ISO/IEC DIS 25436, June 2006.

[69] Bart Jacobs, Rustan Leino, Frank Piessens, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *Proceedings of SEFM*, 2004.

[70] Ghinwa Jalloul. *Concurrent object-oriented systems: a disciplined approach*. PhD thesis, University of Technology, Sydney, Australia, June 1994.

[71] Clifford B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, June 1981.

[72] Clifford B. Jones. *Wanted: a compositional approach to concurrency*, chapter 1, pages 1–15. Springer-Verlag, 2003.

[73] Eric Jul and Bjarne Steensgaard. Implementation of distributed objects in Emerald. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 130–132, Palo Alto, CA, USA, October 1991.

[74] D. G. Kafura and K. H. Lee. Inheritance in actor-based concurrent object-oriented languages. In *European Conference on Object-Oriented Programming*, July 1989.

[75] D. G. Kafura and K. H. Lee. ACT++: Building a concurrent C++ with Actors. *Journal of Object-Oriented Programming (JOOP)*, 3(1):25–37, 1990.

[76] Murat Karaorman and John Bruno. Introduction of concurrency to a sequential language. *Communications of the ACM*, 37(9):103–116, September 1993.

[77] Erol Koç. Concurrent examples in SCOOP. semester thesis, ETH Zurich, Departement Informatik, 2004.

[78] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[79] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition, 1999.

[80] Gary T. Leavens, Erik Poll, C. Clifton, Yonsik Cheon, C. Ruby, D. R. Cok, and Joseph Kiniry. *JML reference manual*. Iowa State University, Department of Computer Science, 2005.

[81] D. J. Lehmann and O. M. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Principles of Programming Languages (POPL)*, pages 133–138, 1981.

[82] Rustan Leino, J. B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Research Report 002, Compaq Research Systems Center, 1999.

[83] R.J. Lipton. Reduction: a method of proving properties of parallel programs. *Communications of the ACM*, 18:717–721, 1975.

[84] Barbara Liskov and Jeannette M. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, November 1994.

[85] K.-P. Löhr. Concurrency annotations. *ACM SIGPLAN Notices*, 27(10):327–340, 1992.

[86] K.-P. Löhr. Concurrency annotations for reusable software. *Communications of the ACM*, 36(9):81–89, 1993.

[87] Formal Methods Europe Ltd. Failures-divergence refinement: FDR2. available online at http://www.formal.demon.co.uk, November–December 2004.

[88] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag, New York, USA, 1995.

[89] Jeremy Manson, William Pugh, and Sarita Adve. The Java memory model. In *Principles of Programming Languages (POPL)*, volume 40 of *SIGPLAN Notices*, pages 378–391, January 2005.

[90] Satoshi Matsuoka and Akinori Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*, chapter Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages, pages 107–150. MIT Press, Cambridge (Mass.), USA, 1993.

[91] Bertrand Meyer. Sequential and concurrent object-oriented programming. In *Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 17–28, Paris, France, June 1990.

[92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, Englewood Cliffs (NJ), USA, March 1992.

[93] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[94] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[95] Bertrand Meyer. The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design. In Tom Lyche Olaf Owe, Stein Krogdahl, editor, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*. Springer Verlag, 2004.

[96] Bertrand Meyer. Attached types and their application to three open problems of object-oriented programming. In *European Conference on Object-Oriented Programming*, pages 1–32, July 2005.

[97] Giuseppe Milicia and Vladimiro Sassone. The inhritance anomaly: ten years later. In *SAC'04*, Nicosia, Cyprus, March 2004.

[98] Jayadev Misra and K. Many Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[99] Richard Mitchell and James McKim. *Design by Contract, by example*. Addison-Wesley, 2001.

[100] Daniel Moser. Design and implementation of a run-time mechanism for deadlock detection in SCOOP. ETH semester project, August 2005. available at http://se.inf.ethz.ch/projects/daniel_moser.

[101] Daniel Moser. Transactions in SCOOP. Master thesis, ETH Zurich, June 2006. available at http://se.inf.ethz.ch/projects/daniel_moser.

[102] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. volume 62, pages 253–286, 2006.

[103] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. Springer Verlag, 2002.

[104] Peter Müller and Joseph N. Ruskiewicz. A modular verification methodology for C# delegates. In Uwe Glässer and Jean-Raymond Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, 2007. Submitted.

[105] Yann Walter Müller. Integrated SCOOP tools. MSc thesis, ETH Zurich, Departement Informatik, February 2007. available at http://se.ethz.ch/research/scoop.

[106] David A. Naumann. Observational purity and encapsulation. In *8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *Lecture Notes in Computer Science*, pages 190–204, April 2005.

[107] Christopher Nenning. Exception handling in SCOOP. Diplomarbeit, ETH Zurich, March 2004.

[108] Piotr Nienaltowski. Efficient data race and deadlock prevention in concurrent object-oriented programs. In *OOPSLA'04 Companion*, pages 56–57, 2004.

[109] Piotr Nienaltowski. Flexible locking in SCOOP. In *International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, pages 71–90, York, United Kingdom, July 2006.

[110] Piotr Nienaltowski. Refined access control policy for SCOOP. Technical Report 511, Computer Science Department, ETH Zurich, February 2006.

[111] Piotr Nienaltowski and Volkan Arslan. SCOOPLI: a library for concurrent object-oriented programming on .NET. In *1st International Workshop on Csharp and .NET Technologies*, Pilsen, Czech Republic, 2003.

[112] Piotr Nienaltowski, Volkan Arslan, and Bertrand Meyer. Concurrent object-oriented programming on .NET. *IEE Proceedings Software, special issue on ROTOR*, 150(8):308–314, October 2003.

[113] Piotr Nienaltowski and Bertrand Meyer. Contracts for concurrency. In *International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, pages 27–49, York, United Kingdom, July 2006.

[114] Jonathan Ostroff, Faraz Ahmadi Torshizi, and Hai Feng Huang. Verifying properties beyond contracts of SCOOP programs. In *International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, pages 4–26, York, UK, July 2006.

[115] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[116] Richard F. Paige and Phillip J. Brooke, editors. *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, number YCS 405. University of York, UK, July 2006.

[117] Michael Papathomas. Concurrency issues in object-oriented programming languages. In Denis Tsichritzis, editor, *Object-Oriented Development*, chapter 12, pages 207–245. University of Geneva, 1989.

[118] Michael Papathomas. *Language design rationale and semantic framework for concurrent object-oriented programming*. Phd thesis, University of Geneva, Switzerland, 1992.

[119] Claude Petitpierre. Synchronous C++, a language for interactive applications. *IEEE Computer*, 31(9):65–72, 1998.

[120] Claude Petitpierre. Synchronous active objects introduce CSP's primitive in Java. In *CAP*, Reading, UK, September 2002.

[121] Gabriel Petrovay. Distributed SCOOP. Master's thesis, University of Cluj-Napoca, Romania, September 2005.

[122] Michael Philippsen. Imperative concurrent object-oriented lanuages: an annotated bibliography. Technical Report TR-95-049, International Computer Science Institute, UCBerkeley, August 1995.

[123] Michael Philippsen. A survey of concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12:917–980, 2000.

[124] Alex Potanin, James Noble, Dave Clarke, and Rober Biddle. Generic Ownership for Generic Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, Oregon, USA, October 2006.

[125] William Pugh. Fixing the Java memory model. In *Java Grande Conference*, pages 89–98, New York, USA, 1999.

[126] Ganesh Ramanathan. Application of SCOOP to control systems. Project report, ETH Zurich, 2007. in preparation.

[127] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2003.

[128] Edwin Rodriguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 551–576, July 2005.

[129] Bill Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[130] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *the Symposium on Operating Systems Principles (SOSP)*, October 1997.

[131] N. R. Scaife. A survey of concurrent object-oriented programming languages. Technical Report RM/96/4, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, February 1996.

[132] Bernd Schoeller and Jonathan Ostroff. Dynamic Contract Frames. *Formal Aspects of Computer Science*, 2007. submitted.

[133] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, January 1993.

[134] Sun Microsystems. *Java Remote Method Invocation (Java RMI)*. available from http://java.sun.com/products/jdk/rmi/.

[135] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), September 2005.

[136] Stanley M. Sutton. Preconditions, postconditions, and provisional execution in software processes. Technical Report UM-CS-1995-077, University of Massachusetts, Amherst, MA, USA, 1995.

[137] S. Tucker Taft, Robert A. Duff, Randall L. Bruckardt, and Erhard Ploederer. *Consolidated Ada reference manual*. LNCS 2219, Springer-Verlag, Berlin, 2000.

[138] Kenjiro Taura, Satoshi Matsuoka, and Akinori Yonezawa. ABCL/f: a future-based polymorphic typed concurrent object-oriented language — its design and implementation. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[139] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with enabled-sets. In *Object-Oriented Programming, Languages and Systems (OOPSLA)*, volume 24 of *103–112*, October 1989.

[140] John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.

[141] Jaco van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, number 2031 in Lecture Notes in Computer Science, pages 299–312, 2001.

[142] Sebastien Vaucouleur and Patrick Eugster. Atomic features. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, CA, USA, October 2005.

[143] Christoph von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD dissertation, ETH Zurich, July 2004.

[144] Christoph von Praun and Thomas Gross. Object-race detection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa Bay, USA, October 2001.

[145] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture*. Prentice-Hall, 1995.

[146] Barbara Wyatt, Krishna Kavi, and Steve Hufnagel. Parallelism in object-oriented languages: a survey. *IEEE Computer*, 11(6):56–66, 1992.

[147] Yasuhiko Yokote and Mario Tokoro. The design and implementation of concurrent Smalltalk. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 331–340, 1986.

[148] Akinori Yonezawa. *ABCL, an Object-Oriented Concurrent System: Theory, Language, Programming, Implementation, and Application*. MIT Press, 1990.

List of Figures

Curriculum Vitae

Piotr Nienaltowski

30 June 1976	Born in Bialystok, Poland
1983 – 1991	Primary school, Czarna Bialostocka, Poland
1991 – 1995	Technical secondary school, Bialystok, Poland
1995	Matura (A-levels) cum laude, Diploma in Automatics and Micromechanics
1995 – 1999	Computer Science studies, Bialystok University of Technology, Poland
1999 – 2000	Computer Science studies, Université Joseph Fourier/INPG Grenoble, France
2000	DEA (MSc) Informatique et Communications, UJF/INPG
2000 – 2002	Research and Teaching Assistant Laboratoire de Téléinformatique, EPFL Lausanne, Switzerland
2002 – 2007	Research and Teaching Assistant Chair of Software Engineering, ETH Zurich, Switzerland