

Practical framework for contract-based concurrent object-oriented programming

Piotr Nienaltowski



Doctoral Thesis ETH No.

# 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 30 June 1976  
Polish citizen

reviewers:  
Prof. Dr. Bertrand Meyer, ETH Zurich, examiner  
Prof. Dr. Peter Müller, ETH Zurich, co-examiner  
Prof. Dr. Jonathan Ostroff, York University, Toronto, Canada, 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. 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 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 minimise the danger of deadlocks.
- A seamless integration of the concurrency model with 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, 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 $\sharp$  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 $\sharp$  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, Faraz Torshizi, Haifeng Huang, Phil Brooke, 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. I have had 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

<b>Abstract</b>	<b>v</b>
<b>Contents</b>	<b>xi</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Thesis statement . . . . .	6
1.2 Goals and evaluation criteria . . . . .	6
1.3 Conventions . . . . .	8
<b>2 Summary and main results</b>	<b>9</b>
2.1 Summary . . . . .	9
2.2 Topics not covered in this dissertation . . . . .	10
2.3 Organisation of the dissertation . . . . .	12
2.4 Main results and contributions . . . . .	12
<b>3 Previous work</b>	<b>25</b>
3.1 Object-oriented concurrency models . . . . .	25
3.2 Multithreading . . . . .	30
3.3 Concurrency in Eiffel . . . . .	35
<b>4 Original SCOOP_97 model</b>	<b>41</b>
4.1 Development . . . . .	41
4.2 SCOOP_97 in detail . . . . .	42
4.2.1 Processors and separate objects . . . . .	42
4.2.2 Separate entities, classes, and calls . . . . .	43
4.2.3 Synchronisation . . . . .	44
4.2.4 Consistency rules . . . . .	48
4.2.5 Additional rules and mechanisms . . . . .	50
4.2.6 Proof rule for feature calls . . . . .	52
4.2.7 Advanced features . . . . .	53
4.3 Related work . . . . .	53
<b>5 Beyond SCOOP_97: critique and roadmap</b>	<b>61</b>
5.1 Semantics of separate annotations . . . . .	61
5.2 Separate call rule . . . . .	63

5.3	Feature call vs. feature application . . . . .	64
5.4	Consistency rules . . . . .	66
5.5	Reasoning about object locality . . . . .	67
5.6	Semantics of contracts . . . . .	68
5.7	Proof rules . . . . .	70
5.8	Locking policy . . . . .	71
5.8.1	Eager locking . . . . .	71
5.8.2	Cross-client locking and separate callbacks . . . . .	72
5.8.3	Void separate arguments . . . . .	73
5.9	Quasi-asynchrony . . . . .	74
5.10	Polymorphism and dynamic binding . . . . .	75
5.11	Genericity . . . . .	77
5.12	Practical considerations . . . . .	78
5.12.1	Enclosing routines . . . . .	78
5.12.2	Deferred classes . . . . .	79
5.12.3	Software reuse . . . . .	80
5.13	Discussion . . . . .	81
<b>6</b>	<b>Type system for SCOOP</b> . . . . .	<b>83</b>
6.1	Computational model . . . . .	83
6.1.1	Feature call . . . . .	84
6.1.2	Feature application . . . . .	86
6.1.3	Synchronisation . . . . .	87
6.2	From consistency rules to a type system . . . . .	88
6.2.1	SCOOP types . . . . .	88
6.2.2	Processor tags . . . . .	90
6.2.3	Implicit types . . . . .	91
6.3	Subtyping . . . . .	92
6.4	Type combinators . . . . .	95
6.5	Valid targets . . . . .	100
6.6	Object creation . . . . .	103
6.7	Handling false traitors . . . . .	104
6.8	Object import . . . . .	106
6.9	Object equality . . . . .	108
6.10	Expanded types . . . . .	109
6.11	Formalisation of the type system . . . . .	111
6.11.1	<i>SCOOP<sub>C</sub></i> programs . . . . .	111
6.11.2	Typing environments . . . . .	112
6.11.3	Valid types . . . . .	114
6.11.4	Subtyping . . . . .	114
6.11.5	Well-formed environments . . . . .	115
6.11.6	Type rules . . . . .	119
6.12	Properties of the type system . . . . .	121
6.12.1	Example . . . . .	121

6.12.2	Lemmas . . . . .	142
<b>7</b>	<b>Flexible locking</b>	<b>145</b>
7.1	Eliminating unnecessary locks . . . . .	145
7.1.1	(Too much) locking considered harmful . . . . .	145
7.1.2	Semantics of attached types . . . . .	146
7.1.3	Support for inheritance and polymorphism . . . . .	147
7.2	Lock passing . . . . .	152
7.2.1	Need for lock passing . . . . .	152
7.2.2	Mechanism . . . . .	153
7.2.3	Lock passing in practice . . . . .	156
7.3	Related work . . . . .	160
<b>8</b>	<b>Contracts and concurrency</b>	<b>163</b>
8.1	Generalised semantics of contracts . . . . .	164
8.1.1	Preconditions . . . . .	164
8.1.2	Postconditions . . . . .	168
8.1.3	Invariants . . . . .	170
8.1.4	Loop assertions and <i>check</i> instructions . . . . .	171
8.2	Towards a proof rule . . . . .	172
8.3	Discussion . . . . .	175
8.3.1	Contract redefinition . . . . .	175
8.3.2	Importance of lock passing . . . . .	176
8.3.3	Run-time assertion checking . . . . .	176
8.4	Related work . . . . .	178
<b>9</b>	<b>Advanced object-oriented mechanisms in SCOOP</b>	<b>181</b>
9.1	Inheritance and polymorphism . . . . .	181
9.1.1	Multiple inheritance . . . . .	181
9.1.2	Polymorphism and dynamic binding . . . . .	183
9.1.3	Deferred classes . . . . .	188
9.2	Genericity . . . . .	189
9.2.1	Generic parameters . . . . .	190
9.2.2	Constrained genericity . . . . .	191
9.2.3	Actual result types . . . . .	193
9.2.4	Actual argument types . . . . .	194
9.2.5	Detachable generic parameters . . . . .	195
9.2.6	Type conformance . . . . .	196
9.2.7	Discussion . . . . .	201
9.3	Agents . . . . .	202
9.3.1	Agents as potential traitors . . . . .	203
9.3.2	Separate agents . . . . .	206
9.3.3	Open targets . . . . .	212
9.3.4	Applications of separate agents . . . . .	213

9.4	Once routines . . . . .	218
9.5	Discussion . . . . .	219
<b>10</b>	<b>Using SCOOP in practice</b>	<b>221</b>
10.1	Classic examples . . . . .	221
10.1.1	Dining philosophers: atomic locking of multiple resources . . . . .	221
10.1.2	Producers-consumers: condition synchronisation . . . . .	225
10.1.3	Binary search trees: efficient parallelisation . . . . .	226
10.1.4	Santa Claus: barriers and priority scheduling . . . . .	229
10.2	Agents and asynchrony . . . . .	232
10.2.1	Rendezvous synchronisation and active objects . . . . .	234
10.2.2	Waiting faster . . . . .	236
10.2.3	Resource pooling . . . . .	239
10.2.4	Event-driven programming . . . . .	242
10.3	Control systems . . . . .	244
10.3.1	Elevator . . . . .	244
10.3.2	Arm robot . . . . .	246
10.4	Code reuse . . . . .	250
10.5	Inheritance anomalies . . . . .	252
10.6	Discussion . . . . .	257
<b>11</b>	<b>Implementation: issues and solutions</b>	<b>259</b>
11.1	Supported mechanisms . . . . .	260
11.2	SCOOPLI library . . . . .	262
11.2.1	Processors . . . . .	263
11.2.2	Separate objects . . . . .	263
11.2.3	Separate calls . . . . .	264
11.2.4	Scheduling, locking, and wait conditions . . . . .	265
11.2.5	Lock passing . . . . .	266
11.2.6	Quiescence and termination . . . . .	266
11.2.7	Garbage collection . . . . .	266
11.3	Scoop2scoopli tool . . . . .	267
11.3.1	Code generation . . . . .	268
11.3.2	Bootstrapping . . . . .	271
11.3.3	Invariant checking . . . . .	272
11.3.4	Postcondition checking . . . . .	272
11.4	CONCURRENCY library . . . . .	273
11.4.1	<i>CONCURRENCY</i> . . . . .	274
11.4.2	<i>EXECUTOR</i> . . . . .	275
11.4.3	<i>ANSWER_COLLECTOR</i> . . . . .	275
11.4.4	<i>EVALUATOR</i> . . . . .	275
11.4.5	<i>POOL_MANAGER</i> . . . . .	276
11.4.6	<i>LOCKER</i> . . . . .	276
11.4.7	SCOOP-enabled <i>EVENT_TYPE</i> . . . . .	276

<b>12 Teaching SCOOP</b>	<b>279</b>
12.1 Topics . . . . .	279
12.2 Assessment . . . . .	280
12.3 Students' feedback . . . . .	281
12.4 Discussion . . . . .	286
<b>13 Critique and conclusions</b>	<b>289</b>
13.1 Applicability to other languages . . . . .	290
13.2 Limitations and future work . . . . .	299
13.3 Final remarks . . . . .	304
<b>A CONCURRENCY library</b>	<b>305</b>
<b>B Glossary</b>	<b>317</b>
<b>Bibliography</b>	<b>321</b>
<b>List of Figures</b>	<b>331</b>



# 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 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 in a consistent state if and 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 clearly correspond 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<sub>97</sub>*, 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<sub>97</sub> 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 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<sub>97</sub> 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 synchronous 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.
- 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 of SCOOP. 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; we simply assume that a violated assertion results in indefinite waiting (see chapter 8). 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-freeness*

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. An attempt at solving this issue is described in [108]. 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<sub>97</sub> 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 rule for proving the correctness of 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 gives a compact summary of the current model and discusses its use in practice. Several examples illustrate the discussion.
- Chapter 11 describes the implementation and the supporting tools; various implementation issues and solutions are discussed.
- Chapter 12 discusses the experience gathered in two iterations of a SCOOP-based graduate course on object-oriented 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<sub>97</sub> model, and continue with a development of current SCOOP. The computational model and the language extension described below are derived from SCOOP<sub>97</sub> 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<sup>1</sup> — 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 (this is enforced by type rules, see section 6.5). Locking is achieved through the refined mechanism of feature application: the handler 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**.
- $x: \text{separate } X$   
Objects attached to  $x$  may be handled by any processor; this processor may (but does not

---

<sup>1</sup>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**  $\langle p \rangle X$   
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**’. ( $y$  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 all 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. 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 handler waits until the precondition is satisfied. A violated precondition results in (possibly infinite) waiting. In practice, the treatment of preconditions is optimised by the run time system to avoid the infinite waiting whenever possible.
- Postcondition 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 be-

cause 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  $(\gamma, \alpha, C)$  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 guaranteed to be non-void at run time; detachable entities may be void [96, 53].
- *Processor tag*  $\alpha \in \{\bullet, \top, p, \perp\}$   
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. ‘ $\perp$ ’ 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

$x: \text{separate } X$

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

$x: \text{?separate } \langle py \rangle X$

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 ‘ $\perp$ ’; 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 subtyping — *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 'T' to something more specific in result types, and in the opposite direction (from more specific to 'T') 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.

$$\frac{\{INV \wedge Pre_r\} \text{ body}_r \{INV \wedge Post_r\}}{\{Pre_r^{ctr}[\bar{a}/\bar{f}]\} \text{ x.r}(\bar{a}) \{Post_r^{ctr}[\bar{a}/\bar{f}]\}} \quad (2.4.1)$$

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<sub>97</sub>. This effort has driven the research and revealed several limitations and inconsistencies within SCOOP<sub>97</sub>, 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 focused 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 <sup>2</sup>. 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.

---

<sup>2</sup>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>