SCOOP – A contract-based concurrent object-oriented programming model

Benjamin Morandi[1], Sebastian S. Bauer[2], and Bertrand Meyer[1]

[1] Chair of Software Engineering, Swiss Federal Institute of Technology Zurich, Switzerland,
`firstname.lastname@inf.ethz.ch`,
`http://se.inf.ethz.ch/`

[2] Institut für Informatik, Ludwig-Maximilians-Universität München, Germany,
`sebastian.bauer@pst.ifi.lmu.de`,
`http://www.pst.ifi.lmu.de/`

Abstract. SCOOP is a concurrent object-oriented programming model based on contracts. The model introduces processors as a new concept and it generalizes existing object-oriented concepts for the concurrent context. Simplicity is the main objective of SCOOP. The model guarantees the absence of data races in any execution of a SCOOP program. This article is a technical description of SCOOP as defined by Nienaltowski [11] and Meyer [7, 9, 10].

1 Introduction

In a semaphore based concurrent programming model it is the responsibility of developers to ensure proper synchronization between threads. With respect to safety, no undesirable interference between threads must occur. In general, this is a global property of a program that requires global analysis. With respect to liveness, every thread should progress eventually. Again, this is a global property. Another issue comes from the limited reusability of classes. A class whose instances should be accessed by multiple threads must be annotated with correct synchronization code. In general, a class that is not annotated accordingly can only be used sequentially.

SCOOP stands for *Simple Concurrent Object-Oriented Programming.* This name captures what SCOOP is all about – a simple object-oriented programming model for concurrency. SCOOP is simple because it introduces only few new concepts on top of an object-oriented programming model. This makes SCOOP simple to understand. SCOOP is simple because it helps to avoid common correctness and liveness issues due to improper synchronization. SCOOP is also simple because a class does not need to be annotated with synchronization code before it can be used in a concurrent program. This fosters reusability. SCOOP is simple because it supports sequential reasoning on concurrent programs. There are many reasons why SCOOP is simple and the following sections will explain in more details where the simplicity is coming from. The risk of simplicity is the loss of expressiveness on account of too many restrictions. SCOOP preserves object-oriented expressivity because it generalizes existing object-oriented concepts in the concurrent context. Sequential programs are treated as a subset of a concurrent programs. As presented here, SCOOP builds on Eiffel [4]; however, it is possible to extend other object-oriented programming models with SCOOP. The key requirements of such an programming model are the presence of contracts and a static type system.

SCOOP has gone through several iterations of refinement. The initial model was proposed by Meyer [7, 9, 10]. Nienaltowski worked on consistency and usability of the model. The following sections provide a concise description of the SCOOP model as presented by Nienaltowski in his PhD dissertation [11]. The next section shows an introductory example to give a brief overview on SCOOP. The remaining sections cover the details. Sections 3, 5 and 6 describe the essence of the computational model. Sections 4, 8, 9, and 10 define the type system. Section 7 describes the impact of SCOOP on contracts and sections 11, 12, and 13 discuss advanced object-oriented mechanisms. We conclude with section 14 on limitations and future work.

2 Example

This section introduces an implementation of a producer-consumer scenario written in SCOOP. The concepts introduced in this example will be explained in depth in the following sections.

A producer-consumer application consists of a number of producers and a number of consumers. Both producers and consumers have access to a shared fixed-size buffer. Producers store elements in the buffer and consumers retrieve elements from the buffer. A producer must not be allowed to add elements to the buffer if it is full. A consumer must not be allowed to remove elements from an empty buffer. The access to the buffer must be restricted to one actor at a time.

Listing 1.1. producer class

```
class PRODUCER[G]

inherit
  PROCESS

create
  make

feature {NONE} -- Initialization
  make (a_buffer: separate BOUNDED_QUEUE[G])
      -- Create a producer with 'a_buffer'.
```

```
    do
        buffer := a_buffer
    ensure
        buffer = a_buffer
    end

feature -- Basic operations
    step
        -- Produce an element and store it in 'buffer'.
    local
        l_element: G
    do
        l_element := ...
        store (buffer, l_element)
    end

feature {NONE} -- Implementation
    buffer: separate BOUNDED_QUEUE[G]
        -- The buffer.

    store (a_buffer: separate BOUNDED_QUEUE[G]; a_element: G)
        -- Store 'a_element' in 'a_buffer'.
    require
        a_buffer_is_not_full: not a_buffer.is_full
    do
        a_buffer.put (a_element)
    ensure
        a_element_is_added: a_buffer.count = old a_buffer.count + 1
    end
end
```

Listing 1.2. consumer class

```
class CONSUMER[G]

inherit
    PROCESS

create
    make

feature {NONE} -- Initialization
    make (a_buffer: separate BOUNDED_QUEUE[G])
        -- Create a consumer with 'a_buffer'.
    do
```

```
      buffer := a_buffer
    ensure
      buffer = a_buffer
    end

feature -- Basic operations
  step
    -- Consume an element from 'buffer'.
  local
    l_element: G
  do
    retrieve (buffer)
    l_element := last_element
    ...
  end

feature {NONE} -- Implementation
  buffer: separate BOUNDED_QUEUE[G]
    -- The buffer.

  retrieve (a_buffer: separate BOUNDED_QUEUE[G])
      -- Retrieve an element from 'a_buffer' and store it in 'last_element'.
    require
      a_buffer_is_not_empty: not a_buffer.is_empty
    do
      last_element := a_buffer.item
      a_buffer.remove
    ensure
      last_element_is_set: last_element = old a_buffer.item
      a_buffer_is_decreased: a_buffer.count = old a_buffer.count - 1
    end

  last_element: G
end
```

Producers and consumers repeatedly perform a sequence of actions. Producers store elements into the buffer and consumers retrieve elements from the buffer. Producers and consumers can therefore be modeled as processes. Both of the classes inherit from a class *PROCESS*, which is not shown here. The class *PROCESS* offers a deferred feature *step*, which gets called over and over again as soon as the feature *live* is called. Therefore the main activities of producers and consumers are enclosed by their respective implementations of *step*.

Both producers and consumers operate on a shared buffer attached to *buffer* . The type **separate** *BOUNDED_QUEUE*[*G*] of these two features is of interest.

The class type *BOUNDED_QUEUE*[*G*] specifies the nature of the buffer: It is a bounded queue with elements of a type *G*. The keyword **separate** is a SCOOP specific addition to the type system. In SCOOP every object is associated to a processor that is responsible for the sequential execution of instructions on that object. One object can only be associated to one processor, but one processor can be responsible for multiple objects. The **separate** keyword defines that the object attached to the entity of such a type can potentially be handled by a different processor than the processor of the current object. In the absence of this keyword the object must be handled by same processor as the current object. If two objects are on different processors then the two processors can execute features on these objects in parallel. In this example, we want to achieve this for the buffer. Features on the buffer should be executed in parallel to the features on producers and consumers. Thus the buffer needs to be on its own processor. This is reflected in the type of the buffer.

The problem description asks for mutual exclusion on the buffer. In SCOOP locks are granted on the granularity level of processors. Locking a processor means exclusive access to all the objects handled by the processor. Prior to its execution, every feature automatically requests the locks on all the processors of the attached formal arguments. The feature cannot be executed until all the requested locks got granted. In the producer and consumer classes it is therefore necessary to enclose the calls to the buffer in features taking the buffer as an attached formal argument in order to satisfy the exclusivity requirement. For this purpose, the producer class has a feature *store* that takes the buffer and the element as formal arguments. A call to *store* gets delayed until the producer acquired the lock on the buffer. Note that the lock on the element is already given because the missing **separate** keyword in the type of the element implies that the element and the producer are on the same processor. Next to the locks there is another constraint that must be satisfied before *store* can be executed. The feature has a precondition asking for the buffer not to be full, as specified. As the buffer is shared among different producers and consumers, the responsibility to satisfy the precondition is also shared among the different producers and consumers. The precondition therefore turns into a wait condition as *store* needs to wait for the precondition to be satisfied. In summary, *store* can only be executed when the precondition is satisfied and the necessary locks are acquired. A similar argument is true for the feature *retrieve* of the consumer. In the next section, we start with an elaboration on the computational model of SCOOP.

3 Processors, objects, and the scheduler

Processors and objects are the basic computational units in SCOOP. A processor is an autonomous thread of control capable of executing features on objects. Every processor is responsible for a set of objects. In this context, a processor is called the handler of the associated objects. Every object is assigned to exactly one processor that is the authority of feature executions on this object. If a processor q wants to call a feature on a target handled by a different processor p then q needs to send a feature request to processor p. This is where the request queue of processor p comes into place. The request queue keeps track of features to be executed on behalf of other processors. Processor q can add a request to this queue and processor p will execute the request as soon as it executed all previous requests in the request queue. Processor p uses its call stack is used to execute the feature request at the beginning of the request queue. Before processor q can add a request, it must have a lock on processor p's request queue. Otherwise another processor could intervene. Once processor q is done with the request queue of processor p it can add an unlock operation to the end of the request queue. This will make sure that the request queue lock of p will be released after all the previous feature requests have been executed. Similarly, processor p must have a lock on its call stack to add features to it. To simplify this, every processor starts with a lock on its own call stack. Section 5 on the semantics of feature calls and feature applications will explain the interaction between processors in more details. In conclusion, a processor and its object form a sequential system. The overall concurrent system can be seen as a collection of interacting sequential systems. A sequential system can be seen as a particular case of a concurrent system with only one processor.

Definition 1 (Processor). *A processor is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. Every processor has the following elements:*

- *Handled objects: It keeps track of the handled objects.*
- *Request queue: It keeps track of features to be executed on objects handled by the processor. Requests in the request queue are serviced in the order of their arrival.*
- *Call stack: It is used for the application of features.*
- *Locks: It contains all the locks held by the processor, as defined in definition 2.*

Definition 2 (Processor locks). *For every processor there exists a lock on the request queue and a lock on the call stack. A lock on the request queue grants permission to add a feature request to the end of the request queue. A lock on the call stack grants permission to add a feature request to the top of the call stack. Initially every processor has a lock on its own call stack and its request queue is not locked.*

Definition 3 (Processor loop). *A processor loops through the following sequence of actions:*

1. *Idle wait: If both the call stack and the request queue are empty then wait for new requests to be enqueued.*
2. *Request scheduling: If the call stack is empty but the request queue is not empty then dequeue an item and push it onto the call stack.*
3. *Request processing: If there is an item on the call stack then pop the item from the call stack and process it.*

- *If the item is a feature request then apply the feature.*
- *If the items is an unlock operation then unlock the request queue of the processor.*

In the standard case, every processor keeps the lock on its own call stack. A processor needs this lock to dequeue items from the request queue and put them on the call stack, as described in definition 3. Normally, only the request queue is used for the communication between different processors. Section 5 will show how this can be different. In the following we will introduce an abstraction under the assumption that every processor keeps its call stack lock. In this abstraction we call the request queue lock on a processor p simply the lock on p. As long as the call stack lock on a processor p is in possession of p, a request queue lock on p in possession of a processor q means that processor p will be executing new feature requests in the request queue exclusively on behalf of q. This means that a request queue lock grants exclusive access to all the objects handled by p. Transferring this insight to our abstractions, a lock on processor p denotes exclusive access to the objects handled by p. We used the abstraction in the beginning of the article, as it is easier to begin with. In the remaining sections we will not make use of this abstraction anymore.

As stated earlier, there is only one processor that can execute features on a particular object. As a consequence, any execution of a SCOOP program is free ob low-level data races that occur when multiple processors access an attribute of an object at the same time and there is at least one write access. Proposition 1 expresses this fact.

Proposition 1. *A SCOOP system is free of low-level data races.*

As mentioned, a request queue can only be accessed by a processor that is in possession of the corresponding request queue lock. The execution of a feature requires access to request queues of other processors. Therefore it is necessary to obtain request queue locks prior to the execution of a feature so that these request queues can be accessed during the execution. There might be more than one processor whose current feature request requires a particular lock. This is where the scheduler comes into place. The scheduler is the arbiter for feature requests. More details on this will be given in section 5. The model permits a number of possible scheduling algorithms. Scheduling algorithms differ in their level of fairness and their performance. In this article we do not focus on a particular instance. More information on particular scheduling algorithms can be found in Nienaltowski's dissertation [11].

Definition 4 (Scheduler). *The scheduler is responsible for scheduling feature applications.*

Processors bring a new dimension to feature calls because feature calls can either happen within one processor or from one processor to another. Thus feature calls can be separate or non-separate depending on the location of the client and the target object.

Definition 5 (Separateness). *Objects that are handled by different processors are separate from each other. All the objects on the same processor are non-separate from each other. A feature call is separate if and only if the target and the client objects are separate. A references to a separate object is called a separate reference.*

4 Types

4.1 Definition

In SCOOP two objects are either separate or non-separate with respect to each other. The separateness depends on the location of the two objects. Throughout the following sections, the relative location of one object to another object will be of importance. Thus there needs to be a way of computing this information. SCOOP uses a refinement of the Eiffel type system to keep track of the relative locations. The Eiffel type system is based on detachable tags and classes. The detachable tag defines whether an entity is allowed to be void or not. In SCOOP the detachable tag has an additional meaning. Section 5 will show that only objects attached to attached entities will be locked. In order to argue about separateness the type system must accommodate the locality of objects in addition to the existing type components. The following definitions refine the definitions in section 8.11 of the Eiffel ECMA standard [4].

Definition 6 (Type). *A type is represented as a triple $T = (d, p, C)$ with the following components:*

- *The component d is the detachable tag as described by definition 7.*
- *The component p is the processor tag as described by definition 8.*
- *The component C is the class type.*

A type is always relative to the instance of the class where the type is declared. An actual generic parameter is always relative to the instance of the class where the corresponding formal generic parameter is declared.

Definition 7 (Detachable tag). *The detachable tag d captures detachability and selective locking.*

- *An entity can be of attached type, formally written as $d = !$. Entities of an attached type are statically guaranteed to be non-void. Only request queues of processors handling arguments of attachable type get locked.*
- *An entity can be of detachable type, formally written as $d = ?$. Entities of detachable type can be void. Request queues of processors handling arguments of detachable type do not get locked.*

Definition 8 (Processor tag). *The processor tag p captures the locality of objects accessed by an entity of type T.*

- *The processor tag p can be separate, formally written as $p = \top$. The object attached to the entity of type T is potentially handled by a different processor than the current processor.*
- *The processor tag p can be explicit, formally written as $p = \alpha$. The object attached to the entity of type T is handled by the processor specified by α. Definition 9 shows how a processor can be specified explicitly.*
- *The processor tag p can be non-separate, formally written as $p = \bullet$. The object attached to the entity of type T is handled by the current processor.*
- *The processor tag p can denote no processor, formally written as $p = \perp$. It is used to type the void reference.*

Note the difference between a separate type and a separate object. A separate object is on a different processor. An entity of a separate type stores a potentially separate object.

Definition 9 (Explicit processor specification). *A processor can be specified explicitly either in an unqualified or a qualified way. An unqualified explicit processor specification is based on a processor attribute p. The processor attribute p must have the type $(!, \bullet, PROCESSOR)$ and it must be declared in the same class as the explicit processor specification or in one of the ancestors. The processor denoted by this explicit processor specification is the processor stored in p. A qualified explicit processor specification relies on an entity e occurring in the same class as the explicit processor specification or in one of the ancestors. The entity e must be a non-writable entity of attached type and the type of e must not have a qualified explicit processor tag. The processor denoted by this explicit processor specification is the same processor as the one of the object referenced by e.*

Explicit processor tags support precise reasoning about object locality. Entities declared with the same processor tag represent objects handled by the same processor. The type system takes advantage of this information to support safe attachments and safe feature calls. A qualified explicit processor specification can only be defined on a non-writable entity of attached type to facilitate static type checking. Possibly void or writable entities would require an analysis of the runtime behavior. This would make static type checking unfeasible. The type of the entity e must not have a qualified explicit processor tag itself. This restriction prevents dependency cycles among processor tags.

4.2 Syntax

SCOOP extends the Eiffel type syntax to incorporate the enhanced type definition.

Definition 10 (Type syntax).

```
type ≜
  [detachable_tag]
  [separate] [explicit_processor_specification]
  class_name [actual_generics]

detachable_tag ≜
  attached | detachable

explicit_processor_specification ≜
  qualified_explicit_processor_specification |
  unqualified_explicit_processor_specification

qualified_explicit_processor_specification ≜
  "<" entity_name "." handler ">"

unqualified_explicit_processor_specification ≜
  "<" entity_name ">"
```

*The absence of both the **attached** and **detachable** keyword implies an attached type.*

The SCOOP syntax change is minimal and ensures backward compatibility to plain Eiffel programs. Definition 10 anticipates a change in the syntax of the detachable tag which is not yet part of the Eiffel ECMA standard [4].

Example 1 (Type syntax). Listing 1.3 shows a couple of attributes along with their types. The comments attached to the attributes explain what is meant by the syntax.

Listing 1.3. type syntax example

```
a: BOOK                                      -- (!, •, BOOK)
b: separate BOOK                             -- (!, ⊤, BOOK)
c: separate <p> BOOK                         -- (!, p, BOOK)
d: attached BOOK                             -- (!, •, BOOK)
e: attached separate BOOK                    -- (!, ⊤, BOOK)
f: attached separate <a.handler> BOOK        -- (!, a.handler, BOOK)
g: detachable BOOK                           -- (?, •, BOOK)
h: detachable separate BOOK                  -- (?, ⊤, BOOK)
i: detachable separate <p> BOOK              -- (?, p, BOOK)

p: PROCESSOR
```

4.3 Explicit and implicit types

Every entity has an explicit type. It is the type as written in the code. Thanks to the qualified explicit processor specifications, every attached and non-writable entity also has an implicit type. This is stated in definition 11. **Current** is one of these attached and non-writable entities and consequently it has an implicit type. The explicit type of **Current** is shown in definition 12. Definition 13 justifies the processor tag $\perp$. It is used to define the type of the void reference. Hereby, the standard Eiffel class *NONE* is used as the class type, because it is at the bottom of the class hierarchy.

Definition 11 (Implicit type). *An attached non-writable entity e of type $(!, p, C)$ also has an implicit type $(!, e.handler, C)$.*

Definition 12 (Explicit type of the current object). *In the context of a class C, the current object has the type $(!, \bullet, C)$.*

Definition 13 (Explicit type of the void reference). *The void reference has the type $(?, \perp, NONE)$.*

4.4 Expanded types

Every object is either of reference type or of expanded type. Instances of classes annotated with the keyword **expanded** are objects of expanded type. Other objects are of reference type. The difference between the two categories affects the semantics of attachment. An attachment of an object of a reference type to an entity stores the reference to the object in the entity. An attachment of an object of expanded type copies the object and attaches it to the entity. Section 7.4 of the Eiffel ECMA standard [4] explains this in more details. Due to the copy semantics, an expanded object is never attached to more than one entity and thus expanded objects do not get aliased. One can think of expanded objects to be part of the object to which they are attached. Thus expanded objects are defined to be non-separate. Furthermore, an entity of expanded type never contains the void reference. Thus an expanded type is always an attached type. This leads to the validity definition 14 for expanded types. Syntactically this rule prohibits the use of separate annotations and the question mark as the detachable tag.

Definition 14 (Expanded type validity). *A type T based on an expanded class E is valid if an only if it is attached and non-separate, i.e. $T = (!, \bullet, E)$.*

4.5 Formal generic parameters

Formal generic parameters are type parameters for generic classes. A generic derivation must be used to get a type from a generic class. In a generic derivation each formal generic parameter must be substituted by a type, which is the actual generic parameter. Optionally, every formal generic parameter can have a constraint on the actual generic parameter used in the generic derivation. Such a constraint allows the generic class to make assumptions on a formal generic parameter. An implicit generic constraint is used if no explicit constraint is given. In the presence of the new type system the implicit constraint as described in section 8.12.7 of the Eiffel ECMA standard [4] must be generalized. For compatibility with the existing rule the implicit constraint must be attached and have the class type ANY. The class ANY is the root class in the Eiffel class hierarchy. An implicit type should be as general as possible. The separate processor tag is most general and thus definition 15 makes use of it.

Definition 15 (Implicit formal generic parameter constraint). *The constraint of a formal generic parameter is the explicit constraint if present. Otherwise the implicit constraint is* $(!, \top, ANY)$.

4.6 Soundness

SCOOP extends the Eiffel type system with information about object locality. This information can be used to determine whether an object is separate or non-separate. To be sound, the type system must ensure that this information is accurate at all times for all entities. In conjunction with the justifications of the rules and mechanisms, the following sections provide arguments on why the type system is sound. One component of soundness is the absence of traitors as defined in definition 16. However, the absence of traitors does not imply full soundness. Soundness must also be guaranteed for types with explicit processor specifications.

Definition 16 (Traitor). *A traitor is an entity declared as non-separate but pointing to a separate object.*

We defer a full soundness proof to later work as described in section 14.

5 Feature call and feature application

A processor p can call features on objects that are either handled by p or by another processor q. A non-separate call is executed by p itself. For a separate call, processor p needs to ask processor q to execute the feature. In this section we will take a closer look at the steps involved in a feature call and in the subsequent execution, which we call the feature application. As we will see, a separate call can be asynchronous, but a non-separate call is always synchronous. Every feature call happens in the context of a feature application. For this reason we will start with the description of the feature application and then describe the feature call. In the end we will present an example to illustrate once again how the two concepts work together. In terms of contracts, this section only describes the runtime aspects of contracts. A more detailed picture will be given in section 7. The definitions presented in this section generalize the definitions in section 8.23 of the Eiffel ECMA standard [4].

5.1 Feature application

We start in a situation where a processor p wants to apply a feature request f on a target x. The execution of f will require a number of request queue locks. Furthermore, the precondition of f must be satisfied before f can be executed. These two prerequisites are established in the synchronization step. This step involves the scheduler. Processor p will wait until the scheduler gives the green light and then execute f. After the execution, the postcondition must be checked. If f is a query then the result must be returned. Finally the obtained request queue locks must be released. Definition 17 captures these steps.

Definition 17 (Feature application). *The application of feature f on target x, requested by a client processor p_c, results in the following sequence of actions performed by the supplier processor p_x:*

1. *Synchronization: Involve the scheduler to wait until the following synchronization conditions are satisfied atomically:*
   - *All the request queues of processors that handle arguments of an attached type in f are locked on behalf of p_x.*
   - *The precondition of f holds.*
2. *Execution*
   - *If f is a routine then run its body.*
   - *If f is an attribute then evaluate it.*
3. *Postcondition evaluation: Every query in the postcondition must be evaluated by its target handler. The result must be combined by p_x if it is involved in the postcondition. Otherwise any involved processor may be used.*
4. *Result returning: If f is a query then return the result to p_c. Results of expanded type need to be imported by the client handler p_c.*
5. *Lock releasing: Add an unlock operation to the end of each request queue that has been locked in the synchronization step.*

Synchronization Before a feature can be applied there are some synchronization constraints to be fulfilled. First, the supplier processor must have atomically acquired all the required request queue locks. The formal argument list of f indicates which request queues must be locked. If a formal argument is declared as attached then the corresponding request queue gets locked. If a formal argument is declared as detachable then the corresponding request queue does not get locked. Note that the feature call rule in definition 22 will show that p_x could already have some locks through a chain of lock passing operations. It is not necessary to reacquire these locks. The selective locking mechanism has advantages over an eager locking mechanism where all the request queues get locked. The likelihood of deadlocks is decreased thanks to fewer locks. Selective locking supports a precise specification of resources needed by a routine and it makes it possible to pass an argument without locking the corresponding request queue. There is a reason why the detachable tag is used to encode selective locking. Assuming a formal argument would be detached then it is not clear how locking should be defined on a detached formal argument. Thus it makes sense to restrict locking to attached formal arguments. This leads to the generalized semantics of the detachable tag. As a second synchronization constraint, the precondition of f must hold. Note that if f is a routine that was called in qualified way and not as a creation procedure then the invariant is part of the precondition, as described in section 8.9.26 of the Eiffel ECMA standard [4]. A violated precondition clause can either cause an exception or it can lead to waiting. Section 7 will show that only unsatisfied controlled precondition clauses cause an exception.

Locking before executing ensures that processor p_x can access the locked request queues without interference caused by other processors. Thus processor p_x can make separate calls without interference as long as all calls are on objects handled by the corresponding processors. There is the assumption that each call stack lock of the argument processors is either in possession of its own processor or in possession of p_x. We will see later why this assumption is always given as we take a look at lock passing. Non-separate calls can also be executed without interference. As we will see, a non-separate call is handled over the call stack of p_x and does not involve the request queue of p_x. A new feature request is simply added to the top of the call stack. No other processor can interfere in this process. In conclusion, there are a number of safe targets which we call controlled. For safety reasons we only allow these targets in feature calls. Definitions 18 and 19 capture this restriction in terms of the type system.

Definition 18 (Controlled expression). *An expression exp of type $T_{exp} = (d, p, C)$ is controlled if and only if exp is attached, i.e. $d = !$, and exp satisfies at least one of the following conditions:*

- *The expression exp is non separate, i.e. $p = \bullet$.*
- *The expression exp appears in a routine r that has an attached formal argument farg with the same handler as exp, i.e. $p = farg.handler$.*

The second condition of definition 18 is satisfied if and only if at least one of the following conditions is true:

- The expression *exp* appears as an attached formal argument of r.
- The expression *exp* has a qualified explicit processor specification *farg.handler* and *farg* is an attached formal argument of r.
- The expression *exp* has an unqualified explicit processor specification p, and some attached formal argument of r has p as its unqualified explicit processor specification.

Definition 19 (Valid target). *Call $exp.f(\overline{a})$ appearing in class C is valid if and only if the following conditions hold:*

- *The expression exp is controlled.*
- *The expression exp's base class has a feature f exported to C, and the actual arguments $\overline{a}$ conform in number and type to the formal arguments of f.*

Definitions 18 and 19 apply to invariants, preconditions, postconditions and the routine bodies of a class. In case of an invariant, there is no enclosing routine. Thus an invariant can only contain non-separate calls. As a further consequence of definition 19. calls on void targets are prohibited. The call validity rule in definition 19 replaces the validity rule in section 8.23.9 of the Eiffel ECMA standard [4]. With this approach some safe feature calls are rejected. Such a situation can occur if there is a feature call on a uncontrolled separate expression to which a non-separate object is attached. In section 11 we will refer to this object as a false traitor.

Example 2 (Valid targets). Listing 1.4 shows a feature *print_book*. This feature makes a copy of the book and notifies the author of the book.

Listing 1.4. printer class

```
class PRINTER feature
  print_book (a_book: separate BOOK)
      -- Print 'a_book'.
    do
      a_book.copy
      a_book.author.notify
    end
end

class BOOK feature
  author: AUTHOR -- The author of this book.
  initial_price: INTEGER -- The price as initially recommended.
  is_clean: BOOLEAN -- Is this book clean?

  copy
      -- Copy this book.
    do
      ...
    ensure
      this_book_is_not_clean: not is_clean
    end

  clean
      -- Clean this book.
    require
      this_book_is_not_clean: not is_clean
    do
      ...
    ensure
      this_book_is_clean: is_clean
    end
```

```
end
```

The author is accessible through a non-separate attribute in the book class. Therefore the author and its book are on the same processor. In this example we want to see whether the two calls *a_book.print* and *a_book.author.notify* are valid according to definition 19. For this, we have to apply definition 18 to the two expressions *a_book* and *a_book.author*. The expression *a_book* is an attached formal argument and therefore it is controlled. The explicit type of *a_book* is $(!, \top, BOOK)$. Judging from this, *a_book* is attached. In addition, *a_book* is non-writable because it is a formal argument. We can therefore use definition 11 to derive the implicit type $(!, a\_book.handler, BOOK)$. We will now use this implicit type to derive the type of the expression *a_book.author*. We noted that the author and the book must be on the same processor. This means that we can use the book's processor tag for the author. The result type combiner in definition 27 will state this more clearly. With the implicit type of *a_book* the type of the expression *a_book.author* becomes $(!, a\_book.handler, AUTHOR)$. In conclusion the expression *a_book.author* has a qualified explicit processor tag that is related to the attached formal argument *a_book*. Therefore the expression is controlled. We can conclude that all targets are valid.

We already argued that any execution of a SCOOP program is free of low-level data races. Next to low-level data races there are high-level data races. They occur if multiple processors access a set of objects in a non-atomic way, i.e. in an interleaved manner, and there is at least one write access. As a consequence of the synchronization step in the feature application rule in definition 17 and the valid call rule in definition 19, a SCOOP system is free of high-level data races by design as stated by proposition 2.

Proposition 2. *A SCOOP system is free of high-level data races.*

Postcondition evaluation The postcondition of the feature must be executed and checked. Note that if f is a routine that was called in qualified way or if f was called as a creation procedure then the invariant is part of this postcondition, as described in section 8.9.26 of the Eiffel ECMA standard [4]. There is one obvious way to evaluate the postcondition. Every processor that handles one or more targets in the postcondition evaluates its share of the postcondition while processor p_x is waiting for the results to come back. Once all the results have been gathered, processor p_x can determine whether the postcondition holds or not. However, this approach introduces sequentiality because p_x must wait for the query evaluations to finish. It turns out that p_x does not need to wait if none of its objects is used as a target in the postcondition. Any involved processor will do.

In any case, processor p_x gets the guarantee that the postcondition will be satisfied eventually. More precisely, it will be satisfied when the execution of the features called within f terminated. The last set of feature requests on any postcondition target will be the ones coming from the postcondition evaluation. Thus the postcondition gets evaluated at the right time just after all the feature called within f have terminated. Further feature request on the postcondition targets, issued after the execution of f, will only be applied after the postcondition has been evaluated. The postcondition might not hold right after the execution of f; but it will will hold when it becomes relevant, just before other feature request can be issued on the postcondition targets.

A postcondition evaluation can result in a violated postcondition. Such a state will trigger an exception in the supplier processor.

Result returning We consider a situation where f returns an expanded object. The copy semantics of expanded objects could mislead to return a shallow copy from p_x to p_c. This is not a good idea. If the result object on p_x has an attached non-separate entity then the copied object on p_c has a non-separate entity to which a separate object on p_x is attached. We would have introduced a traitor. We introduce the import operation as defined in definition 20 to return an expanded object without this issue.

Definition 20 (Import operation). *The import operation executed by a processor p and applied to an object o on a different processor q involves the following sequence of actions:*

1. *Make a copy of o called o'.*
2. *Make a copy of all the non-copied objects that are reachable from o through non-separate references.*
3. *For every non-separate once function f of every copied object the following actions must be done:*
   (a) *If f is fresh on p and non-fresh on q then f must be marked as non-fresh on p and the value of f on q must be used as the value of f on p.*
   (b) *If f is fresh on p and fresh on q then f remains fresh on p.*
   (c) *If f is non-fresh on p then f remains non-fresh on p and the result of f on p remains the same.*
4. *For every once procedure f of every copied object the following actions must be done:*
   (a) *If f is fresh on p and non-fresh on q then f must be marked as non-fresh on p.*
   (b) *If f is non-fresh on p then f remains non-fresh on p.*
5. *Rewire the copied object structure in such a way that it mirrors the original object structure. Separate entities do not need to be altered in the copied object structure as they can still point to the original separate objects.*
6. *Put the copied object structure on p.*

Objects reachable through separate references do not need to be copied as their entities are already equipped with the right type with respect to p. Once functions with a non-separate result type complicate the import operation a bit. As it will be formulated later in definition 37, such a function f must be evaluated at most once on a given processor p and all the objects on p with f must share the result. We need to be careful in a situation where we import an object with a function f that has already been evaluated on processor q and on processor p. We cannot have two different values for the same once function on the same processor. Definition 20 takes care of issues of this kind. Similarly, it takes care of once procedures; they have a once per processor semantics as well. The terminology on freshness has been taken from section 8.23.20 of the ECMA standard [4]. A once routine is called fresh on a particular processor if and only if it has never been executed on any object handled by this processor. Otherwise the once routine is called non-fresh.

Example 3 (Import operation). Figure 1 shows the objects o, a, b and c forming an object structure. The objects o and a are on the same processor. The objects b and c are on separate processors. The result of the import operation applied to the object o by a processor p different than the handler of o is shown in the lower half of figure 1. Objects o results in a copied object o' on p. Because object a is non-separate with respect to o, processor p receives a copied object a' as well. The objects b and c do not need to be copied. They can be shared by both object structures as they were separate.

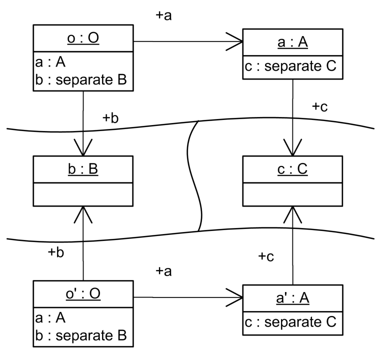

Fig. 1. import operation example object diagram

The import operation computes the non-separate version of an object structure. It potentially results in a copied object structure that contains both copied and original objects. This can be an issue in case one of the copied objects has an invariant over the identities of objects as example 4 shows.

Example 4 (Invariant violation as a result of the import operation). Imagine two objects x and y on one processor and another object z on another processor. Object x has a separate reference a to z and a non-separate reference b to y.

Object z has a separate reference c to y. Object x has an invariant with a query $a.c = b$. An import operation on x executed by a third processor will result in two new objects x' and y' on the third processor. The reference a of object x' will point to the original z. The reference b of object x' will point to the new object y'. Now object x' is inconsistent, because $a.c$ and b identify different objects, namely y and y'.

The deep import operation is a variant of the import operation that does not mix the copied and the original objects. The drawback of the deep import operation is the increased overhead.

Definition 21 (Deep import operation). *The deep import operation executed by a processor p and applied to an object o on a different processor involves the following sequence of actions:*

1. *Make a copy of o called o'.*
2. *Make a copy ons' of all the non-copied objects that are reachable from o through non-separate references.*
3. *Make a copy os' of all the non-copied objects that are reachable from o through separate references.*
4. *For every non-separate once function f of every copied object the following actions must be done:*
   - *(a) If f is fresh on p and non-fresh on q then f must be marked as non-fresh on p and the value of f on q must be used as the value of f on p.*
   - *(b) If f is fresh on p and fresh on q then f remains fresh on p.*
   - *(c) If f is non-fresh on p then f remains non-fresh on p and the result of f on p remains the same.*
5. *For every once procedure f of every copied object the following actions must be done:*
   - *(a) If f is fresh on p and non-fresh on q then f must be marked as non-fresh on p.*
   - *(b) If f is non-fresh on p then f remains non-fresh on p.*
6. *Rewire the copied object structure in such a way that it mirrors the original object structure.*
7. *Put the copied object o' and the non-separate copied object structure ons' on p.*
8. *Put each copied object in the separate copied object structure os' on the processor of the respective original object.*

Example 5 (Deep import operation). Figure 2 shows the objects o, a, b and c forming an object structure. The objects o and a are on the same processor. The objects b and c are on separate processors. The result of the deep import operation applied on the object o by a processor p different than the handler of o is shown on the lower half of the figure. All the objects involved in the object structure got copied.

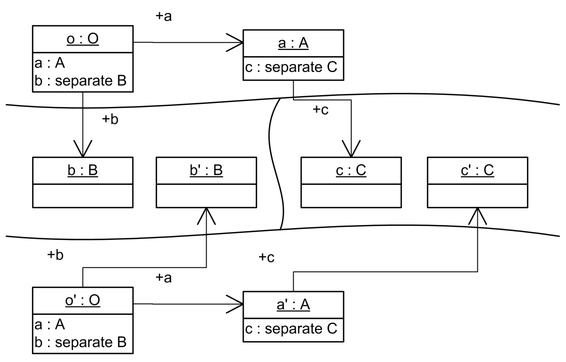

Fig. 2. deep import operation example object diagram

Lock releasing After the execution of f, processor p_x does not require the request queue locks anymore. At this point the request queue locks might still contain feature requests from p_x. However, there are no more new feature request because the execution of f is over. Therefore p_x can release the locks. For this, processor p_x does not wait until all the features requests triggered in the execution of f finished. Instead, it appends an unlock operation to each locked request queue. As a result, different request queues may become unlocked at different times after they are done with all the feature requests. Next to an increase in parallelism, asynchronous unlocking permits safe realization of certain synchronization scenarios that would lead to deadlocks otherwise.

5.2 Feature call

So far we studied the context in which a feature call can occur. A feature call results in the generation of feature request by the client processor on a potentially different supplier processor. The feature request gets executed eventually by the supplier processor. Note the change of roles. A supplier processor becomes a client processor when the supplier processor makes a call.

The calling processor has a number of locks to ensure exclusive access. The client processor might be in possession of some locks that are necessary in the synchronization step of the resulting feature request on the supplier processor. In such a situation, the client processor can temporarily pass its locks to the supplier processor. This step is called lock passing and it happens right after argument passing. Once these two steps completed, the client processor can place its feature request. If the called feature is a query then the client processor must wait for the result. Before the feature call can be completed, the client processor must revoke the passed locks. Definition 22 explains these steps in more details.

Definition 22 (Feature call). *A feature call $x.f(\overline{a})$ results in the following sequence of actions performed by the client processor p_c:*

1. *Argument passing:*
   - *Attach the actual arguments $\overline{a}$ to the formal arguments of f.*
   - *Arguments of expanded type need to be imported by the supplier processor p_x.*
2. *Lock passing:*
   - *Full lock passing: If any controlled actual argument of a reference type gets attached to an attached formal argument of f then pass all the currently held request queue locks and call stack locks to p_x.*
   - *Call stack locks passing: In case there are no such arguments and there is a separate callback, i.e. p_x already had a request queue lock on p_c at the moment of the call then pass the call stack locks to p_x.*
3. *Feature request: Ask p_x to apply f to x.*
   - *Schedule f for an immediate execution by p_x using the call stack of p_x and wait until it terminates, if any of the following conditions hold:*
     - *The feature call is non-separate, i.e. $p_c = p_x$.*
     - *The feature call is a separate callback.*
   - *Otherwise, schedule f to execute after the previous calls on p_x using the request queue of p_x.*
4. *Wait by necessity: If f is a query then wait for its result.*
5. *Lock revocation: If lock passing happened then wait for f to terminate and revoke the locks.*

Lock passing, feature request, and lock revocation Deadlock avoidance is the motivation behind lock passing. Without lock passing, it is very easy to get into a deadlock. Suppose that processor p_x needs a particular lock in the application of feature f. If this lock is in possession of processor p_c then p_x cannot proceed until p_c releases this lock. If f happens to be a query then p_c has to wait for the result of f while p_c is holding on to the lock. According to the conditions given by Coffman et al [3] a deadlock occurred. If the client processor p_c could temporarily pass on the lock to p_x then p_x would be able to apply the requested feature, return the result, and let p_c continue. We call this solution full lock passing.

Full lock passing includes passing the request queue locks and the call stack locks. It happens if any controlled actual argument of a reference type gets attached to an attached formal argument of f. An attached formal argument means that the request queue will be locked in the synchronization step of f's application. A controlled actual argument means that p_c has a request queue lock on the handler of the actual argument. In short, p_c has a lock required by p_x. Full lock passing is only relevant for arguments of a reference type. Arguments of expanded type will be copied using the import operation during argument passing.

If full lock passing happens then p_c passes all its locks to p_x and not only the locks that triggered the lock passing mechanism. This generous behavior eliminates more potential deadlocks compared to a solution where only a subset of the locks gets passed. As soon as at least one lock gets passed, processor p_c cannot proceed until it can revoke the locks after the feature f terminated. Otherwise the atomicity guarantees expressed by proposition 2 could be violated due to two processor who could potentially be working on the same set of objects. As p_c has to wait in any case that involves lock passing, it does not hurt to pass all the locks and not just a subset.

A feature call results in a feature request. The client processor p_c generates a feature request to be handled by the supplier processor p_x. The feature request could either go to the end of the request queue or on top of the call stack of processor p_x. If the two processors are the same then the call is non-separate and f must be applied immediately. This is the sequential case. The request queue is irrelevant for this and only the call stack is involved. There is another, more special case, where feature f must be applied immediately. A separate callback occurs when the supplier processor p_x already held a lock on the client processor p_c at the moment of the feature call to f. This can happen in the following situation: We assume p_x initiated a chain of feature calls involving full lock passing. At the end of this chain processor p_c executes the feature call $x.f(\overline{a})$. Processor p_x is responsible for the application of this call. At this point processor p_x is waiting until its feature call terminates to revoke the locks. We assume the feature call f involves lock passing as well. In this situation p_c will wait for the feature call f to terminate. If the feature call f gets added to the end of the request queue of p_x then the system ends up in a deadlock. Processor p_c would be waiting for processor p_x to finish the feature application $x.f(\overline{a})$. But processor p_x would never get to this point because it would still be waiting for its locks to come back. Immediate execution is the way out of this. The feature f must be applied immediately using the call stack of p_x. At the moment of a separate callback the processor p_c is in possession of the call stack lock on p_x, because p_x passed its locks. However, p_x will require this lock during its own execution. Therefore p_c must temporarily pass back its call stack locks to p_x and wait for the locks to return. Again, it does not hurt to pass back all call stack locks instead of just one. In the remaining case the call is a normal separate call. The feature request gets added to the end of the request queue of processor p_x. The processor loop described in definition 3 will put the feature request on top of the call stack as soon as all previous feature requests terminated.

Note that it is not possible to lock a request queue of a processor p that is not in possession of its own call stack lock. We assume p is not in possession of its own call stack lock. This is only possible if p passed its locks in a feature call. This means that p is currently applying a feature and waiting for the locks to return. While p is applying a feature, its request queue is locked. Therefore it is not possible to lock the request queue of p. If a processor q has a request queue lock on p then there are two options. Processor q could have acquired the request queue lock in the synchronization step and therefore p is in possession of its call stack lock. The request queue lock on p could also have been passed to

q. This means that some processor must have acquired the request queue lock on p. The only way how p could have lost its call stack lock is a chain of lock passing operation where processor p is involved. In this case p would have passed on its call stack lock to processor q.

Wait by necessity Next to lock passing, there is another situation where p_c has to wait for p_x to finish its feature application of f. If f is a query then the subsequent statements after the call to f potentially depend on the result of the query. Thus processor p_c needs to wait for the query to terminate before it can proceed. This scheme is called wait by necessity.

5.3 Lock revocation

After the feature f terminated, the locks of p_x and p_c must be restored to the state before lock passing happened.

Valid assumptions after a feature call Processor p_c can be assured that every future feature call on a target mentioned in the postcondition of f will be applied after all the feature requests resulting from the application of f. This includes the feature requests resulting from a postcondition evaluation. This is ensured by the synchronization step in the application of f.

Example 6 (Feature calls and feature applications). Consider the feature *sell_book* in listing 1.5.

Listing 1.5. seller class

```
class SELLER feature
  sell_book (a_book: separate BOOK; a_buyer: separate BUYER; a_valuer:
      separate VALUER)
      -- Sell 'a_book' to 'a_buyer' after asking 'a_valuer' for the price.
    local
      l_estimated_price: INTEGER
    do
      a_book.clean
      l_estimated_price := a_valuer.estimate (a_book)
      a_buyer.buy (a_book, l_estimated_price)
    end
end

class VALUER feature
  estimate (a_book: separate BOOK): INTEGER
      -- The estimated price of 'a_book'.
    do
      Result := f (a_book.initial_price)
    end
end
```

We use the following notation to describe a processor p with a request queue rq, request queue locks rql, and call stack locks csl: $p :: rq,\ rql,\ csl$. We start from a point where the request queue of the current processor p_c contains the feature *sell_book*.

p_c :: (**Current**.*sell_book* (*a_book*, *a_buyer*, *a_valuer*)), (), (p_c)
p_{book} :: (), (), (p_{book})
p_{valuer} :: (), (), (p_{valuer})

As a first step, p_c removes the feature *sell_book* from its request queue and puts it on its call stack as described in definition 3. Next p_c starts with the feature application according to definition 17. As there is no precondition, processor p_c asks the scheduler to get the request queue locks on the handlers p_{book}, p_{buyer}, and p_{valuer}. We assume that each of these handlers are different from each other. Eventually these locks are available and p_c can execute the body of *sell_book*. Note that *sell_book* is now on the call stack and not in the request queue anymore.

$p_c :: (),\ (p_{book}, p_{buyer}, p_{valuer}),\ (p_c)$
$p_{book} :: (),\ (),\ (p_{book})$
$p_{valuer} :: (),\ (),\ (p_{valuer})$

The body has three feature calls. Their semantics is described in definition 22. The treatment of *a_book.clean* is easy. There are no arguments to be passed. The feature request step results in the following situation:

$p_c :: (),\ (p_{book}, p_{buyer}, p_{valuer}),\ (p_c)$
p_{book} :: (*a_book.clean*), (), (p_{book})
$p_{valuer} :: (),\ (),\ (p_{valuer})$

The remaining two steps of a feature call do not apply here. The treatment of *a_valuer.estimate* (*a_book*) is more complex as it involves lock passing. According to definition 18 the expression *a_book* is controlled in the feature *sell_book*. The expression is used as an actual argument of reference type in the call. The corresponding formal argument is attached. We just encountered a situation where the caller has a request queue lock which is necessary in the execution of the supplier. Lock passing and the addition of a feature request result in the following situation:

$p_c :: (),\ (),\ ()$
p_{book} :: (*a_book.clean*), (), (p_{book})
p_{valuer} :: (*a_valuer.estimate* (*a_book*)), (p_{book}, p_{buyer}, p_{valuer}), (p_{valuer}, p_c)

Note that the call stack lock of p_c gets passed to give p_{valuer} a chance to handle a separate callback. In the current example we do not make use of this. At this point processor p_c has to wait until the locks can be revoked. While p_c is waiting, processors p_{book} and p_{valuer} proceed in parallel. They can dequeue a feature from the beginning of their request queues, put it on their call stacks, and apply the features.

p_c :: (), (), ()
p_{book} :: (), (), (p_{book})
p_{valuer} :: (), (p_{book}, p_{buyer}, p_{valuer}), (p_{valuer}, p_c)

At this point p_c can retrieve the result, revoke the locks and do the assignment.

p_c :: (), (p_{book}, p_{buyer}, p_{valuer}), (p_c)
p_{book} :: (), (), (p_{book})
p_{valuer} :: (), (), (p_{valuer})

The last instruction *a_buyer.buy* (*a_book*, *l_estimated_price*) triggers another passing of locks. Here, processor p_c will have to wait due to the passed locks, even though the instruction itself does not impose wait by necessity. Last but not least, p_c will add unlock operations to the end of the request queues of p_{book}, p_{buyer}, and p_{valuer}.

6 Object creation

Constructing objects is more complicated in SCOOP than in Eiffel because an object needs to be created on a processor. Definition 23 refines the definitions in section 8.20 of the Eiffel ECMA standard [4].

Definition 23 (Object creation). *A creation call $x.cp(\overline{a})$ on the target x of type (d, p, C) and with the creation procedure cp results in the following sequence of actions performed by the client processor p_c:*

1. *Processor creation*
   - *If x is separate, i.e. $p = \top$, then create a new processor p_x.*
   - *If x has an explicit processor specification, i.e. $p = \alpha$, then*
     - *if the processor denoted by p already exists then take $p_x = p_p$.*
     - *if the processor denoted by p does not exist yet then create a new processor p_x.*
   - *If x is non-separate, i.e. $p = \bullet$, then take $p_x = p_c$.*
2. *Locking: If $p_x \neq p_c$ and p_c does not have a request queue lock on p_x yet then lock the request queue of p_x.*
3. *Object creation: Ask p_x to create a fresh instance of C using the creation procedure cp. Attach the newly created object to x.*

4. *Lock releasing: Add an unlock operation to the request queue of p_x if a lock has been obtained earlier.*

The type of an entity specifies the locality of the attached object. When an entity is used as the target of a creation routine then the new object must be handled by a compatible processor. In some cases such a processor might already exist in other cases a compatible processor must be created. If $p = \bullet$ then the new object must be created on the current processor. If $p = \top$ then any processor could be taken. To exploit parallelism, a new processor gets created. An explicit processor specification specifies a particular processor. If the explicit processor specification is qualified then the specified processor exist already because there is an attached entity whose object is handled by this processor. For an unqualified explicit processor specification it might be necessary to create a new processor if this did not happen already.

After the new object got created there needs to be a call to the creation routine. This call is handled like a feature call as described in definition 22. If the call to the creation routine is separate and p_c does not have the request queue lock on p_x then it is necessary to acquire a request queue lock on p_x before the call. The lock must be released after the call. The new object gets attached to the entity x as soon as the object is created but without waiting for the creation procedure to be applied. This means that x points to a potentially inconsistent object until the creation procedure terminates. However, this is not harmful because new feature requests will be added after the feature request for the creation routine.

Example 7 (Object creation). Feature *initialize* in listing 1.6 shows four creation instructions for four different books. In this example we will go through this list and explain what will happen at runtime.

Listing 1.6. book collection class

```
class BOOK_COLLECTION feature
  hamlet: HAMLET -- Hamlet.
  robinson: separate ROBINSON -- Robinson.
  cinderella: separate <p> CINDERELLA – Cinderella.
  tarzan: separate <p> TARZAN -- Tarzan.

  p: PROCESSOR

  initialize
      -- Initialize this book collection.
    do
      create hamlet
      create robinson
      create cinderella
      create tarzan
    end
```

```
end

class HAMLET inherit BOOK end
class ROBINSON inherit BOOK end
class CINDERELLA inherit BOOK end
class TARZAN inherit BOOK end
```

The first instruction creates a book and stores it in *hamlet*. The type of this entity is non-separate. Thus Hamlet will be created on the current processor. The second instruction creates the book called Robinson. The type of the entity is separate and thus a new processor must be created and used as the handler of the new book. The third instruction creates another classic called Cinderella. The type **separate** <*p*> *CINDERELLA* has an unqualified explicit processor specification. We assume that the specified processor has not been created before. Under this assumption, the processor must be created and used as the handler of the new book. In the last instruction the book called Tarzan gets created. The type of the target **separate** <*p*> *TARZAN* has an unqualified explicit processor specification that specifies the same processor as the entity *cinderella*. Based on the previous instruction it is clear that this processor already exists. Thus there is no need to create a new processor. The books Cinderella and Tarzan are handled by the same processor.

7 Contracts

Design by Contract [8] introduces a new paradigm in object-oriented programming. The use of contracts imposes a crucial reduction of complexities in object-oriented development, in particular when it comes to correctness reasoning. By enriching class interfaces with contracts each class implementation can be verified and proven correct separately. Contracts typically consist of preconditions and postconditions for features and invariants on a class level. These contracts result in mutual obligations of clients and suppliers. In the context of classes enriched with contracts, the principle called separation of concerns gains in importance because the client can rely on the interface of the supplier without the need to know its implementation details. Eiffel supports contracts in the form of assertions being checked at runtime. Sections 7.5 and 8.9 from the Eiffel standard [4] provide more details on contracts in Eiffel.

Unfortunately the traditional interpretation of contracts breaks down in the context of concurrency. In concurrent programs a client calling a feature of a class generally cannot establish the precondition of the feature any more. The reason is that in general feature calls are asynchronous and the point in time of the feature call and the moment of the actual execution of the feature body do not coincide, as it is the case in a sequential program. Thus the objects involved in the precondition may be changed in between by feature calls from other clients. This results in the situation where the precondition that was satisfied at the moment of the call is violated at the moment of execution. Similarly, postconditions cannot be interpreted as full guarantees any more.

SCOOP introduces a new approach to a uniform and clear semantics of contracts in a concurrent context. Thus SCOOP generalizes the principles of Design by Contract, and additionally fosters the use of modular proof techniques. The advantage of the proposed semantics of contracts is that it applies equally well in concurrent and sequential contexts. Following the idea of Eiffel, contracts in SCOOP are formulated as assertions directly written into the code and evaluated during runtime. If an assertion is evaluated and it is not satisfied, then an exception is raised. For preconditions this rule must be carefully revisited due to the observation made above that it may happen that a caller (from a different handler than the target object's handler) cannot be held responsible for establishing the whole assertion of the feature. These considerations result in a refined rule saying that a violated precondition clause that is not under the control of the caller does not lead to an exception - instead the feature call is queued for a later application. Similarly, the semantics of postconditions are adapted to the concurrent context.

7.1 Controlled and uncontrolled assertion clauses

Following the new generalized semantics of contracts proposed by [11], the handling of a feature call strongly depends on the controllability of the involved assertion clauses. The notion of controlled and uncontrolled assertion clauses introduced in the following essentially captures the idea of controlled expressions (definition 18): An assertion clause is called controlled with respect to the current context if all involved objects are under the control and cannot be modified by other processors. Otherwise the assertion clause is called uncontrolled.

Definition 24 (Controlled assertion 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.

Example 8 (Controlled and uncontrolled precondition clauses). We illustrate the difference between controlled and uncontrolled precondition clauses by the example shown in listing 1.7.

Listing 1.7. cleaner class

```
class CLEANER feature
  manual: BOOK -- The cleaning manual.

  clean (a_book: separate BOOK)
       -- Clean 'a_book'.
```

```
    require
      a_book_is_not_clean: not a_book.is_clean
    do
      a_book.clean
    end
end

class CLEAN_BOOK_COLLECTION inherit BOOK_COLLECTION feature
  clean_all (a_cleaner: separate CLEANER; a_extra_book: separate BOOK)
      -- Clean all available books.
    require
      ...
    do
      -- Clean all books in the collection.
      a_cleaner.clean (robinson) -- a_book_is_not_clean uncontrolled
      a_cleaner.clean (hamlet) -- a_book_is_not_clean controlled
      ...
      -- Clean additional books.
      a_cleaner.clean (a_extra_book) -- a_book_is_not_clean controlled
      a_cleaner.clean (a_cleaner.manual) -- a_book_is_not_clean controlled
    end
end
```

We consider a client calling the feature *clean_all*. In the body of *clean_all*, the precondition *a_book_is_not_clean* of the feature call *a_cleaner.clean* (*robinson*) is uncontrolled since *robinson* is not controlled in the context of *clean_all*; *robinson* is declared as a potentially separate object whose processor's request queue is not locked in *clean_all*. On the other hand, *a_book_is_not_clean* is controlled in the three remaining calls to *clean* because the targets of the call to *is_clean* in the precondition are controlled in *clean_all*. The expression *hamlet* is non-separate hence controlled. The expression *a_extra_book* is separate but it is a formal argument of *clean_all*. Therefore it also controlled. Finally, *a_cleaner.manual* is separate in the context of *clean_all*, but it is non-separate from *a_cleaner* and *a_cleaner* is controlled hence *a_cleaner.manual* is controlled too.

Remark 1. The notion of an assertion clause originates in section 8.9 of the Eiffel ECMA standard [4].

7.2 Semantics of contracts

In the following we will precisely describe how contracts given by invariants, preconditions and postconditions are interpreted during runtime of a SCOOP program.

Semantics of preconditions In concurrent programs the usual correctness semantics of preconditions does not fit anymore because in general the client cannot guarantee that the precondition will hold at the moment of the feature application. This inconsistency in the standard interpretation of preconditions in the concurrent context is called *separate precondition paradox* in [11]. This suggests the *wait semantics* for preconditions involving separate clauses. If the precondition is violated only due to violated uncontrolled precondition clauses, the feature application has to be delayed until the precondition clauses holds. On the other hand, a violated controlled precondition clause has to be blamed on the client because no other processor than the client's processor could have accessed the objects occurring in a controlled precondition clause. For such a case an exception needs to be raised in the client. Asynchronous Exceptions raise some problems; this is discussed in section 14.

Example 9 (Precondition semantics). We consider class *READING_LIST* in listing 1.8. It used a bounded buffer to maintain books to be read.

Listing 1.8. reading list class

```
class READING_LIST inherit BOOK_COLLECTION feature
  bestsellers: separate BUFFER[separate BOOK] -- The bestsellers.
  favorites: BUFFER[separate BOOK] -- The favorites.

  store (a_book_list: separate BUFFER[separate BOOK]; a_book: BOOK)
      -- Store 'a_book' in 'a_book_list'.
    require
      a_book_list_is_not_full: not a_book_list.is_full
      a_book_is_clean: a_book.is_clean
    do
      a_book_list.put (a_book)
    ensure
      a_book_list_is_not_empty: not a_book_list.is_empty
    end

  get (a_book_list: separate BUFFER[separate BOOK]): separate BOOK
      -- Remove a book from 'a_book_list'.
    require
      a_book_list_is_not_empty: not a_book_list.is_empty
    do
      Result := a_book_list.get
    end

  add_hamlet_to_all (a_extra_book_list: separate BUFFER[separate BOOK])
      -- Add Hamlet to all book lists including 'a_extra_book_list'.
    require
      ...
    do
      store (a_extra_book_list, hamlet)
      store (bestsellers, hamlet)
      store (favorites, hamlet)
    end
end
```

The feature *store* has as formal arguments a book list and a book; when applied it puts the book into the book list. The precondition of that feature requires that the book list is not full and moreover, that the book is clean. The latter is always a correctness condition since waiting is meaningless if the book is not clean. However, the semantics of the former precondition depends on the locality of the actual arguments. This is illustrated by the feature *add_hamlet_to_all*, where there are three feature calls to *store*. For the first call the precondition *a_book_list_is_not_full* is a correctness condition since *a_extra_book_list* is controlled and hence the precondition clause is controlled. For the second call the precondition is a waiting condition since *bestsellers* is uncontrolled. Finally, for the third call the precondition is a correctness condition since *favorites* is a non-separate attribute of the class and hence *a_book_list_is_not_full* is controlled as well.

Definition 25 (Precondition semantics). *A precondition expresses the necessary requirements for a correct application of the feature. The execution of the feature's body is delayed until the uncontrolled precondition clauses are satisfied. A violated controlled precondition clause immediately marks the precondition as violated.*

The generalized semantics proposed in [11, 12] comprises both interpretations of precondition clauses. As seen in the example, they can be correctness conditions or wait conditions. Correctness conditions only apply to those clauses that are controlled by the client: the client can ensure that the precondition clause hold at the moment of the feature application. The uncontrolled precondition clauses cannot be established by the client, i.e., the client cannot take the responsibility for satisfying them at the moment of the feature application. For this reason wait semantics are applied in this case. Note that waiting always happens at the supplier side. Wait conditions can be used to synchronize processors with each other. A supplier processor only proceeds when the wait condition is established.

Semantics of postconditions Similar to the previously mentioned separate precondition paradox, we can constitute a *separate postcondition paradox* for postconditions. On return from a separate call, the client cannot be sure that the postcondition still holds. After the termination of the call and *before* returning from the call another object may have modified the state of an object occurring in an uncontrolled postcondition clause. However, the client knows that the postcondition was fulfilled on termination of the call. Thus after returning from the call the client can only assume the controlled postcondition clauses since no other client can invalidate these. The interpretation of postconditions is symmetric to the treatment of preconditions. Controlled postcondition clauses are a guarantee given to the client and an obligation on the supplier.

In order to avoid blocking semantics of postconditions and to increase parallelism, postconditions are evaluated individually and asynchronously by the object's handler. This means that the client can continue its own activity after returning from a feature call without waiting for the evaluation of a postcondition. The client gets the guarantee that the postcondition will hold eventually.

Example 10 (Postcondition semantics). Listing 1.9 shows a testable version of class *READING_LIST*.

Listing 1.9. reading list test class

```
class TESTABLE_READING_LIST inherit READING_LIST feature
  test (a_extra_book_list: separate BUFFER[separate BOOK])
      -- Run a test on all book lists including 'a_extra_book_list'.
    require
      ...
    local
      l_book: separate BOOK
    do
      store (a_extra_book_list, hamlet)
      store (bestsellers, hamlet)
      store (favorites, hamlet)

      l_book := get (a_extra_book_list)
      l_book := get (bestsellers)
      l_book := get (favorites)
    end
end
```

The feature call *store* (*a_extra_book_list*, *hamlet*) in feature *test* has a controlled postcondition. The postcondition involves an asynchronous call to the separate entity *a_extra_book_list*. However, the postcondition can be assumed immediately, because it will hold eventually. The second (again asynchronous) call *store* (*bestsellers*, *hamlet*) ensures the uncontrolled postcondition. The caller gets the guarantee that the postcondition holds after termination but the postcondition cannot be assumed at a later point in time since the current processor does not have a request queue lock on *bestsellers*. For the call *get* (*a_extra_book_list*), the precondition is controlled, hence it is a correctness condition and it holds since the postcondition of *store* (*a_extra_book_list*, *hamlet*) can be assumed. For the second call *get* (*bestsellers*), the precondition is uncontrolled, hence it is a waiting condition. The postcondition of *store* (*bestsellers*, *hamlet*) can be assumed to hold on termination of that feature, but not at the time of the call *get* (*bestsellers*).

Definition 26 (Postcondition semantics). *A postcondition describes the result of a feature's application. Postconditions are evaluated asynchronously; wait by necessity (i.e. the need to wait for a result of the feature application) does not apply. Postcondition clauses that do not involve calls on objects handled by the same processors are evaluated independently.*

A violation of a postcondition clause raises an exception in the processor that has evaluated this clause.

Semantics of invariants Invariants express class level consistency conditions on objects that must be satisfied in every observable state (see sections 7.5 and 8.9.16 of the ECMA standard [4]). This means that invariants must be satisfied before and after every generally or selectively exported routine that is not used as a creation procedure. In case of a routine used as a creation procedure the invariant must be satisfied after the execution. On the evaluation side invariants get evaluated on both start and termination of a qualified call to a routine that is not used as a creation procedure. It is also evaluated after every call to a creation procedure (see 8.9.26 of the ECMA standard [4]). Invariants are allowed to have non-separate calls only - separate calls are prohibited. This is a direct consequence of the target validity rule 19. Therefore they can be evaluated without the acquisition of further locks. Note that a feature used in an invariant can have separate formal arguments.

Semantics of loop assertions and check instructions There are further types of assertions namely loop variants, loop invariants, and check instructions. Similar to the semantics of postconditions they are evaluated asynchronously, hence wait by necessity does not apply here. Because the assertions cannot be split up in individual clauses (see remark above) the assertion is evaluated at once. Formal reasoning is again not affected since they can (like postconditions) be assumed immediately. Notice that all such assertions are controlled since all call targets must be controlled. If a loop assertion or a check fails, an exception is raised in the supplier.

7.3 Proof rule

The new generalized semantics of contracts in a concurrent context suggest the following mutual obligations between clients and suppliers. The supplier may assume all the controlled and uncontrolled precondition clauses and must ensure - after the execution of the routine's body - all controlled and uncontrolled postcondition clauses. These obligations are exactly the same as in a sequential context, thus from the contract point of view, the same implementation is suitable for both sequential and concurrent contexts. However, in the concurrent context the obligations and the guarantees of the client differ. The client must establish all *controlled* precondition clauses. The uncontrolled precondition clauses will possibly delay the execution of the feature due to the wait semantics, but nevertheless they will hold when the execution of the feature's body takes place. Conversely, the client can only assume the *controlled* postcondition clauses, because - even though the supplier must establish all postcondition clauses - in the meantime uncontrolled objects involved in an uncontrolled postcondition clause may have changed. Hence the client has fewer obligations but it gets fewer guarantees. This is expressed in the following proof rule.

$$\frac{\{INV \wedge Pre_r\}body_r\{INV \wedge Post_r\}}{\{Pre_r^{ctr}[\overline{a}/\overline{f}]\}x.r(\overline{a})\{Post_r^{ctr}[\overline{a}/\overline{f}]\}} \tag{1}$$

$Pre_r^{ctr}[\overline{a}/\overline{f}]$ denotes the controlled clauses of the precondition of the routine r with the formal arguments $\overline{f}$ substituted simultaneously by $\overline{a}$, similarly for $Post_r^{ctr}[\overline{a}/\overline{f}]$. With this proof rule we can prove partial correctness of routines. Given that under the assumption $INV \wedge Pre_r$ the executing of $body_r$ results in a state where $INV \wedge Post_r$ holds, we can deduce that in a given context the call $x.r(\overline{a})$ in a state where $Pre_r^{ctr}[\overline{a}/\overline{f}]$ is satisfied leads to a state where $Post_r^{ctr}[\overline{a}/\overline{f}]$ holds. This proof rule is parametrized by the context. The resulting precondition and postcondition clauses depend on the context, which is expressed in the conclusion by adding *ctr* to the precondition and postcondition.

With the new proof rule we cannot prove total correctness, what we can prove however is partial correctness. Uncontrolled preconditions and postconditions can lead to deadlocks and infinite waiting on non-satisfied preconditions. In its current state the programming model of SCOOP cannot rule out deadlocks completely. However, the likelihood of a deadlock is decreased significantly by introducing selective locking and lock passing. See the outlook section 14 for future work and work that has been done to improve that fact. As pointed out in [11], a fully modular proof system for SCOOP would require much more expressive contracts.

The new proof rule looks very similar to the sequential Hoare rule [6]. The difference between the Hoare rule and the new proof rule resides in the conclusion. The new proof rule limits the assertion clauses to controlled assertion clauses. There is however a case where the Hoare rule becomes a special case of the new proof rule. In a sequential program every assertion clause that involves only attached entities is controlled. Therefore if all assertion clauses only involve attached entities then every assertion clause becomes controlled, i.e. $Pre_r[\overline{a}/\overline{f}] = Pre_r^{ctr}[\overline{a}/\overline{f}]$ and $Post_r[\overline{a}/\overline{f}] = Post_r^{ctr}[\overline{a}/\overline{f}]$.

8 Type combiners

An entity e declared as non-separate is seen as such by the current object o. However, separate clients of o should see e as separate because from their point of view the object attached to e is handled by a different processor. Following this thought, there is a need to determine how a particular type is perceived from the point of view of an object different than the current object. Type combiners are the answer to this question.

8.1 Result type combiner

The result type combiner shown in definition 27 determines the type T_e of a query call $x.f$ based on the type T_{target} of x and the type T_{result} of f. The result type combiner gives the result type of a query from the perspective of the client. The type T_{result} is relative to the target x and the result type combiner determines how this type should be seen by the client.

Definition 27 (Result type combiner). $* : \mathit{Type} \times \mathit{Type} \mapsto \mathit{Type}$

$$(d_1, p_1, C_1) * (d_2, p_2, C_2) = \begin{cases} (!, \bullet, C_2) & \mathit{if\ isExpanded}(C_2) \\ (d_2, p_1, C_2) & \mathit{if} \neg \mathit{isExpanded}(C_2) \wedge p_2 = \bullet \\ (d_2, \top, C_2) & \mathit{otherwise} \end{cases}$$

The result type combiner is a function of two arguments. The first argument is the type of the target T_{target} and the second argument is the type of the result T_{result}.

The first case handles the situation where the result class type is expanded. Results of expanded types are passed back to the client using the import operation described in definition 20. Doing so the result becomes non-separate from the perspective of the client. Thus the result type combiner yields non-separateness as the combined type. The result stays expanded and thus the combined type must be attached. The remaining cases handle the situations where the class of the return type is not expanded.

If the result type is non-separate with respect to the target, i.e. $p_2 = \bullet$, then we conclude that the result must be handled by the same processor as the target. Therefore the combined type has the processor tag of the target type. This situation is handled by the second case.

If the result type is separate with respect to the target, i.e. $p_2 = \top$, then the result can be considered separate from the point of view of the client. This works because $p = \top$ means potentially separate. Thus the combined type can be separate as well. This is reflected in case number three.

If the result type explicitly denotes a processor, i.e. $p_2 = \alpha$, then one could think that the processor tag of the combined type must be p_2 because it is an exact specification of a processor. This is not true. The explicit processor tag p_2 only makes sense in the context of class C_2 for the target x. A processor tag is not a global identification. However, the client can conclude that the result will be potentially separate. This is shown in the third case.

Example 11 (Basic usage of the result type combiner). In combination with genericity the result type combiner can get complicated. Consider listing 1.10.

Listing 1.10. simple library class

```
class LIST[G -> separate ANY] feature
  last: G
      -- The last element of the list.

  put (a_element: G)
      -- Add 'a_element' to the list.
    do
      ...
    end
end

class SIMPLE_LIBRARY feature
  books: LIST[separate BOOK] -- The books.
end
```

The class *SIMPLE_LIBRARY* declares a feature *books* of type *LIST*[**separate** *BOOK*]. The actual generic parameter **separate** *BOOK* is relative to the object attached to *books*. The result type combiner determines the type of *books.last* from the perspective of the library. The type of the target *books* is given by $(!, \bullet, LIST[(!, \top, BOOK)])$. The result type of *last* is $(!, \top, BOOK)$. As a result one gets $(!, \bullet, LIST[(!, \top, BOOK)]) * (!, \top, BOOK) = (!, \top, BOOK)$.

Example 12 (Iterative usage of the result type combiner). The result type combiner can be applied iteratively to multi-dot expressions. Consider listing 1.11.

Listing 1.11. stacked library class

```
class STACK[G] feature
  top: G -- The top element.
end

class STACKED_LIBRARY feature
  books: LIST[STACK[separate BOOK]] -- The books.
end
```

The class *STACKED_LIBRARY* defines a feature *books* of type *LIST*[*STACK* [**separate** *BOOK*]]. In this example we will determine the combined type of *books.last.top* from the perspective of an instance of *STACKED_LIBRARY*. The result type combiner must be applied from left to right because the targets are determined from left to right. The target type of *books* together with the result type of *last* result in the first combined type. This first combined type is the target type for the call to *top*. This target type and the result type of *top* result in the final combined type.

$$(!,\bullet,\mathit{LIST}[B]) * \overbrace{(!,\bullet,\mathit{STACK}[A])}^{B} * \overbrace{(!,\top,\mathit{BOOK})}^{A} =$$
$$(!,\bullet,\mathit{STACK}[A]) * \overbrace{(!,\top,\mathit{BOOK})}^{A} = (!,\top,\mathit{BOOK})$$

8.2 Argument type combiner

The argument type combiner determines the admissible type T_{actual} of an actual argument a in a call $x.f(a)$. It is based on the target type T_{target} and the type T_{formal} of the formal argument. In other words the argument type combiner determines how the client perceives the type of an argument.

Definition 28 (Argument type combiner). $\otimes : \mathit{Type} \times \mathit{Type} \mapsto \mathit{Type}$

$$(d_1, p_1, C_1) \otimes (d_2, p_2, C_2) = \begin{cases} (!,\bullet,C_2) & \mathit{if}\ \mathit{isExpanded}(C_2) \\ (d_2,p_1,C_2) & \mathit{if}\ \neg\mathit{isExpanded}(C_2) \wedge p_1 \neq \top \wedge p_2 = \bullet \\ (d_2,\top,C_2) & \mathit{if}\ \neg\mathit{isExpanded}(C_2) \wedge p_2 = \top \\ (d_2,\bot,C_2) & \mathit{otherwise} \end{cases}$$

The argument type combiner is a function of two arguments. The first argument T_{target} is the type of the target and the second argument T_{formal} is the type of the formal argument.

The first case handles formal arguments of expanded type. Actual arguments of expanded types are passed to the supplier using the import operation described in definition 20. Doing so, the actual argument becomes non-separate from the perspective of the supplier. The client can assume the argument is non-separate. Therefore the argument type combiner yields non-separateness as the combined type. The actual argument is expanded and thus the combined type needs to be attached. The remaining cases handle the situations where the class of the actual argument type is not expanded.

If the formal argument type is non-separate with respect to the target, i.e. $p_2 = \bullet$, then we know that the actual argument must be handled by the same processor as the target. This processor is specified by the target type. If the target type is separate, i.e. $p_1 = \top$, then there is no chance of knowing which processor it is. In the remaining cases we know with certainty which processor to use for the actual argument: when the target type explicitly denotes a processor, i.e. $p_1 = \alpha$, when the target type is non-separate, i.e. $p_1 = \bullet$, or when $p_1 = \bot$. The situation where $p_1 = \bot$ cannot occur because this processor tag is only used to type the void reference. In conclusion, we can only know which processor is expected if $p_1 \neq \top$. If this condition is satisfied then the combined type can have the processor tag of the target type. This scenario is described in the second case.

If the formal argument type is separate relative to the target, i.e. $p_2 = \top$ then the client can provide an actual argument on any processor. Therefore the actual argument can be considered as potentially separate from the perspective of the client. This scenario is handled by the third case.

If the formal argument type explicitly names a processor, i.e. $p_2 = \alpha$, then one could think that the processor tag of the combined type must be the processor tag of the formal argument type because we can exactly determine the processor of the actual argument. This is not true. The processor tag is not a global identification. It only makes sense in the context of class C_2 for the target x. In this situation we know that f is expecting an actual argument on a particular processor, but we do not know which one. Therefore this situation is illegal. This is indicated in the forth case where the processor tag of the combined type is set to $\bot$. The forth case also handles the situation where the formal argument is non-separate with respect to the target, i.e. $p_2 = \bullet$ but the target type is separate, i.e. $p_1 = \top$. As explained earlier this situation is illegal as well.

9 Type conformance

In this section we will refine the existing type conformance rules described in sections 8.14.6 and 8.14.8 of the Eiffel ECMA standard [4] for the new type system to ensure soundness. We define the conformance of one type to another type over the conformance of the type components. Definition 29 states this more clearly. We use the symbol $\sqsubseteq$ for class type conformance and we use the symbol $\preceq$ for type conformance. The typing environment Γ contains the class hierarchy of the program enriched with *ANY* and *NONE* along with the type declaration for all features, local variables, and formal arguments as defined by Nienaltowski [11].

Definition 29 (Type conformance).

$$\frac{\begin{array}{l}\Gamma \vdash E_1 \sqsubseteq E_2 \\ \Gamma \vdash \forall j \in \{1, \dots, m\}, (d_t, p_t, C_t) = \mathit{relatedActualGenericParameter}(b_j) : (\\ \quad (d_t, p_t, C_t) \preceq (d_{b_j}, p_{b_j}, C_{b_j}) \wedge \\ \quad ((d_t = d_{b_j} = ?) \vee (d_t = d_{b_j} \wedge p_t = p_{b_j} \wedge C_t = C_{b_j})) \\ )\end{array}}{\Gamma \vdash E_1[a_1, \dots, a_n] \sqsubseteq E_2[b_1, \dots, b_m = (d_{b_m}, p_{b_m}, C_{b_m})]} \tag{2}$$

The related actual generic parameter of an actual generic parameter b_j is the actual generic parameter a_i whose formal generic parameter is used in the inheritance declaration of E_1 as an actual generic parameter for the formal generic parameter of b_j, provided such an a_i exists. Otherwise it is the actual generic parameter for the formal generic parameter of b_j as defined in the inheritance declaration of E_1 or one of its ancestors.

$$\frac{\begin{array}{l}\Gamma \vdash C_1 \sqsubseteq C_2 \\ \Gamma \vdash \mathit{isExpanded}(C_2) \rightarrow (C_1 = C_2)\end{array}}{\Gamma \vdash (d, p, C_1) \preceq (d, p, C_2)} \tag{3}$$

$$\frac{\Gamma \vdash (d, p, C_1) \preceq (d, p, C_2)}{\Gamma \vdash (d, p_1, C_1) \preceq (d, \top, C_2)} \qquad \frac{\Gamma \vdash (d, p, C_1) \preceq (d, p, C_2)}{\Gamma \vdash (d, \bot, C_1) \preceq (d, p_2, C_2)} \tag{4}$$

$$\frac{\Gamma \vdash (d, p_1, C_1) \preceq (d, p_2, C_2)}{\Gamma \vdash (!, p_1, C_1) \preceq (?, p_2, C_2)} \tag{5}$$

Example 13 (Related actual generic parameters). Listing 1.12 shows the class *ARRAY*, which inherits from class *INDEXABLE*.

Listing 1.12. array and indexable classes

```
class ARRAY[F] inherit INDEXABLE[INTEGER, F] ... end

class INDEXABLE[G, H] ... end
```

We use E_1 to identify the type *ARRAY*[**separate** *BOOK*] and we use E_2 to identify the type *INDEXABLE*[*INTEGER*, **separate** *BOOK*]. We use a_1 for the single actual generic parameter in E_1 and we use b_1 and b_2 to denote the first and the second actual generic parameters in E_2. The goal of this example is to find the related actual generic parameters of b_1 and b_2. The formal generic parameter of a_1 is *F*. In the inheritance declaration of class *ARRAY* the formal generic parameter *F* is used as an actual generic parameter for the formal generic parameter *H* of class *INDEXABLE*. As b_2 belongs to *H*, a_1 is the related actual generic parameter of b_2. For b_1 there are no more actual generic parameter in E_1 that could serve as the related actual generic parameter. However, class *ARRAY* uses $(!, \bullet, \mathit{INTEGER})$ as the actual generic parameter for the formal generic parameter *G* of class *INDEXABLE*. As b_1 belongs to *G*, $(!, \bullet, \mathit{INTEGER})$ is the related actual generic parameter of b_1.

Equations 2 and 3 deal with class type conformance. Equation 2 deals with generically derived class types and equation 3 handles class types that are not generically derived. In principle, equation 2 is the covariant Eiffel rule with a restriction that prevents traitors as a special form of cat calls. Such a cat call is shown in example 14. To prevent catcalls, the definition requires equality between two related actual generic parameters. This requirement can only be ignored if the actual generic parameter in the sub type is detachable. This implies that the corresponding formal generic parameter has a detachable constraint. As a consequence, every feature that has a formal argument of a type equal to such a detachable formal generic parameter must ensure that the formal argument is non-void prior to a safe call. The object test is a mechanism to test whether an expression is non-void. In addition, an object test ensures that the attached object has a certain dynamic type. Note that the dynamic type includes the processor tag. In conclusion, a detachable actual generic parameter implies the necessity of a check of the processor tag. A detachable actual generic parameter in the sub type implies a detachable actual generic parameter in the super type because the sub type must conform to the super type. More information on object tests can be taken from the Eiffel ECMA standard [4]. Equation 3 shows that expanded classes cannot serve as ancestors and thus a class type conforms to an expanded class type only if the two class types are actually the same.

The processor tag conformance rule in equation 4 states that every processor tag conforms to the $\top$ processor tag. Furthermore it defines that the $\bot$ processor tag conforms to every other processor tag. As a result, processor tags can be arranged in a lattice with the $\top$ processor tag on the top and the $\bot$ processor tag at the bottom. Every other processor tag is in the middle, conforming to the top element. The bottom element conforms directly to the middle elements and indirectly to the top element. The $\top$ processor tag denotes a potentially separate processor. An object on any processor can be attached to an entity of such a type. Therefore the explicit processor tag and the non-separate processor tag conform to the $\top$ processor tag. The $\bot$ processor tag symbolizes no processor and it is used to type the void reference. A void reference can be assigned to any writable entity of detachable type, regardless of the processor tag of the entity. As a consequence, the $\bot$ processor tag conforms to any other processor tag. Note that the explicit processor tag does not conform to the non-separate processor tag, even though one can denote the current processor with the explicit processor tag.

An entity of detachable type potentially has an object attached to it. Equation 5 states that the ! detachable tag conforms to the ? detachable tag. The reverse argument is not true. An entity of attached type cannot store a void reference. Note that this definition is compatible with the self-initialization rule for generic parameters as described in section 8.12.6 of the Eiffel ECMA standard [4].

Example 14 (Traitor cat calls). In listing 1.13 the class *ILLEGAL_LIBRARY* declares an attribute *books* of type *LIST*[**separate** *BOOK*].

Listing 1.13. illegal library class

```
class ILLEGAL_LIBRARY feature
  initialize
      -- Initialize this library.
    do
      create {LIST[BOOK]} books
      books.put (create {separate BOOK})
    end

  books: LIST[separate BOOK] -- The books.
end
```

The type of the formal argument in *books.put* is **separate** *BOOK*. Therefore the feature *books.put* can be called with **create** {**separate** *BOOK*} as an actual argument. If equation 2 would permit covariant actual generic parameters without restrictions then it would be possible to attach an object of type *LIST*[*BOOK*] to the entity *books*. However, a call to the feature *books.put* would then result in a traitor, because the object stored in *books* expects a non-separate formal argument whereas the call provides a separate actual argument. For this reason definition 29 does not allow the attachment of an object of type *LIST*[*BOOK*] to an entity of type *LIST*[**separate** *BOOK*].

Definition 29 implies that there must be a root type in the type system. Any object can be attached to an entity of this type. In the Eiffel type system, the class *ANY* is at the top of the type hierarchy. Thus *ANY* is suitable as the class type component of the root type. To be most general, the root type must be detachable and separate.

Definition 30 (Root type). *The root type is* $(?, \top, ANY)$*.*

Example 15 (Valid and invalid subtypes). Listing 1.14 shows a number of entities. In this example we will explore whether these entities can be assigned to each other.

Listing 1.14. entities to demonstrate valid and invalid subtypes

```
a: HAMLET
b: detachable separate BOOK
c: separate <p> BOOK
d: separate <q> BOOK
e: ARRAY[detachable HAMLET]
f: ARRAY[HAMLET]
g: INDEXABLE[INTEGER, detachable separate BOOK]

p: PROCESSOR
q: PROCESSOR
```

We will start with the entities a and b. We will use definition 29 to determine whether $(!, \bullet, HAMLET)$ conforms to $(?, \top, BOOK)$. We omit premises that do not apply and we omit premises that are satisfied trivially.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash HAMLET \sqsubseteq BOOK}}{\Gamma \vdash (d, p, HAMLET) \preceq (d, p, BOOK)}}{\Gamma \vdash (d, \bullet, HAMLET) \preceq (d, \top, BOOK)}}{\Gamma \vdash (!, \bullet, HAMLET) \preceq (?, \top, BOOK)}$$

We read the derivation bottom-up. In the first step we use the detachable tag conformance rule from equation 5. In the second step we use the processor tag conformance rule from equation 4. The class type conformance rule from equation 3 leads us to the last premise, which can be derived from the typing environment. The details on the typing environments can be taken from section 6.11.4 in Nienaltowski's dissertation [11]. The derivation shows that a can be assigned to b. In a similar way, one can derive that c and d can be assigned to b. It is however not possible to do any other assignments among a, b, c, and d. In particular, c cannot be assigned to d because the types specify different processors.

So far we only looked at types that are not generically derived. In a next step we will take a look at generically derived types to see whether e can be assigned to g. We use the class type conformance rule for generically derived class types from equation 2.

$$\frac{\dfrac{}{\mathit{\Gamma} \vdash \mathit{ARRAY} \sqsubseteq \mathit{INDEXABLE}} \quad \dfrac{\dfrac{\dfrac{}{\mathit{\Gamma} \vdash \mathit{HAMLET} \sqsubseteq \mathit{BOOK}}}{\mathit{\Gamma} \vdash (d, p, \mathit{HAMLET}) \preceq (d, p, \mathit{BOOK})}}{\mathit{\Gamma} \vdash (?, \bullet, \mathit{HAMLET}) \preceq (?, \top, \mathit{BOOK})}}{\mathit{\Gamma} \vdash \mathit{ARRAY}[(?, \bullet, \mathit{HAMLET})] \sqsubseteq \mathit{INDEXABLE}[(!, \bullet, \mathit{INTEGER}), (?, \top, \mathit{BOOK})]}$$

We do not show the premise $(!, \bullet, \mathit{INTEGER}) \preceq (!, \bullet, \mathit{INTEGER})$, because it is satisfied trivially. In the same spirit we do not show the premise $(d_t = d_{b_j} = ?) \vee (d_t = d_{b_j} \wedge p_t = p_{b_j} \wedge C_t = C_{b_j})$. The derivation shows that indeed e can be assigned to g. The entity f cannot be assigned to g. This is due to the attached actual generic parameter of f, which is not compatible with the detachable generic parameter in g.

10 Feature redeclaration

A child class inherits features from a parent class. An inherited feature can either be reused, undefined, or redeclared. In a redeclaration, the child class provides a new implementation. The redeclaration can have a weaker precondition and it can have a stronger postcondition. Any feature redeclaration must ensure that the redeclared version of the feature can be called whenever the parent feature can be called. In particular, the contracts and the signatures must be compatible. Sections 8.10.26, 8.14.4, and 8.14.5 of the Eiffel ECMA standard [4] define rules to take care of this for Eiffel. In this section we will refine these rules for SCOOP. Definition 31 defines valid result type redeclarations and definition 32 does the same for formal arguments.

Definition 31 (Valid result type redeclaration). *The result type of a feature can be redeclared from T_1 to T_2 if and only if T_2 conforms to T_1, i.e. $T_2 \preceq T_1$.*

Just like in Eiffel, the result type can be redeclared covariantly. For all three components of a SCOOP type it is always possible to return something more specific than what the client of a feature expects.

Definition 32 (Valid formal argument redeclaration). *The type of a formal argument x can be redeclared from $T_1 = (d_1, p_1, C_1)$ to $T_2 = (d_2, p_2, C_2)$ if and only if all of the following conditions are true:*

- *If T_1 is detachable then T_2 is detachable, i.e. $d_1 = ? \rightarrow d_2 = ?$. T_2 can only be detachable if x is not a target in the inherited postcondition.*
- *Types T_2 and T_1 have identical processor tags, i.e. $p_2 = p_1$, or T_2 is separate, i.e. $p_2 = \top$.*
- *Class type C_2 conforms to C_1, i.e. $C_2 \sqsubseteq C_1$. If C_2 and C_1 are not the same then T_2 is detachable, i.e. $C_2 \neq C_1 \rightarrow d_2 = ?$.*

In Eiffel, formal arguments can be redeclared in a covariant way. However, if the class type changes then the redeclared formal argument must be detachable. A detachable formal argument can contain the void reference. This forces the redeclared feature to use an object test to ensure that the formal argument is non-void. Next to the non-void check the object test ensures that the formal argument has a certain dynamic type. Therefore the redeclared feature is required to check the dynamic type of the formal argument. This makes it possible for the redeclared feature to receive an actual argument whose type is a super type of the redeclared formal argument type, as it is possible in a covariant redeclaration. Definition 32 goes along this line for the class type. A class type of a formal argument can be redeclared covariantly as long as the redeclared formal argument becomes detachable. The processor tag of a formal argument can be redeclared contravariantly. The covariant redeclaration is not allowed for processor tags because it would lead to traitors. If the processor tag of a formal argument can be redeclared covariantly then it would be possible to redeclare a separate formal argument to non-separate. The contravariant redeclaration is not a problem because the redeclared feature can always use a more general processor tag.

Detachable tags encode selective locking. Assuming a formal argument could be redeclared covariantly from detachable to attached then the application of the redeclared feature would lock the request queue of the formal argument. However, the parent feature specifies a non-locking formal argument. The redeclared feature could not be called whenever the parent feature is called. On the other hand a formal argument can be redeclared contravariantly from attached to detachable because this would alleviate the locking requirements. Furthermore it is always safe to assume a detachable formal argument when the actual argument is non-void.

A redeclaration of a formal argument from attached to detachable imposes a risk on the validity of the inherited postcondition. Assuming a parent feature has a postcondition clause that contains a query on a formal argument. According to the valid target rule in definition 19 this formal argument must be attached. A redeclaration of the formal argument from attached to detachable renders the inherited postcondition clause invalid. An invalid postcondition clause is equivalent to a weaker postcondition and thus this situation is not acceptable. Therefore a formal argument can only be redeclared from attached to detachable if the formal argument is not a target in the inherited postcondition clause.

There is a similar issue for inherited precondition clauses. A redeclaration of a formal argument from attached to detachable renders the precondition clause invalid. An invalid precondition clause is equivalent to a weaker precondition. This situation is accepted because this is only a problem for the redeclaring feature and not for the client of the feature. The redeclared feature can assume a weaker precondition as it ignores the invalid precondition clause. As a consequence, such a precondition clause can be assumed to hold vacuously. This is expressed in definition 33.

Definition 33 (Inherited precondition rule). *Inherited precondition clauses with calls on a detachable formal argument hold vacuously.*

Example 16 (Valid feature redeclaration). Listing 1.15 shows a valid redeclaration of the feature *cheaper_alternative*.

Listing 1.15. finder class

```
class LOCAL_FINDER feature
  cheaper_alternative (a_book: BOOK): BOOK
      -- A cheaper alternative to 'a_book'.
    do
      ...
    ensure
      Result.initial_price < a_book.initial_price
    end
end

class WORLDWIDE_FINDER

inherit LOCAL_FINDER
  redefine
    cheaper_alternative
  end

feature
  cheaper_alternative (a_book: separate BOOK): BOOK
      -- A cheaper alternative to 'a_book'.
    do
      ...
    end
end
```

The formal argument gets redeclared from $FT_1 = (!, \bullet, BOOK)$ to $FT_2 = (!, \top, BOOK)$. This is valid according to definition 32. Note that FT_2 cannot be detachable because the formal argument is a target in the inherited postcondition. A detachable type would make the inherited postcondition invalid. The processor tag changes from non-separate to separate. It is allowed to accept a non-separate object in a separate entity.

11 False traitors

At runtime a non-separate object can get attached to a separate entity. The type system permits this. The downside of such an action is a loss of information in the type system. We know that the entity points to a non-separate object, but the type system cannot make this assumption. For example it is not possible to assign the separate entity to a non-separate entity. The type system would complain about a traitor, even though the attached object is in fact non-separate. We call such an object a false traitor.

Definition 34 (False traitor). *A false traitor is a non-separate object accessible through to a separate expression.*

This is not a SCOOP specific problem. The same issue occurs when an object gets attached to an entity whose static type is a proper parent of the object's dynamic type. The solution is the same as in Eiffel. An object test can be used to ensure that the dynamic type of the expression is non-separate.

12 Agents

Agent objects wrap feature calls. An agent can be passed around and the wrapper feature can be called at a later time. When the agent gets created any of the actual arguments can be predefined. An agent call must only provide the remaining actual arguments. These remaining actual arguments are called open and the predefined ones are called closed. Similarly, it is possible to have an open or a closed target. An open target specifies the type of the future target instead of the target itself. In this section we will discuss the location of a new agent. We consider two options. The agent could be handled by the current processor or the agent could be handled by the processor of the target.

The creation of a new agent on the current processor causes problems. Such an agent would be non-separate from the current object. Therefore the agent would always be a valid target on the current processor. If the agent encapsulates a feature call on a separate target then the current processor could call the encapsulated feature on the separate object without having acquired a request queue lock. The agent would be a non-separate object that encapsulates a separate object and we would have a traitor situation.

If the new agent is handled by the same processor as the target then this problem does not occur. This way, the agent represents its target properly in terms of location. Agent calls can be freely mixed with other feature calls. A lock on the request queue of the handler of encapsulated target is ensured through a lock on the request queue of the handler of the agent. There is however a price for this scheme with respect to open targets. At construction time, the handler of the agent must be known and it must be equal to the handler of the future target. If the target type is non-separate then this is not a problem because the exact processor is know. If the target type has an explicit processor specification then the exact processor is known at creation time. However, the explicit processor specification is only valid in the context where the agent gets created. If the agent gets called in a different context then the exact processor of the target is unknown at call time. If the target type is separate then there is no way of knowing the exact handler when the agent gets created. In conclusion, the type of an open target must be non-separate. As a further restriction, the open target type must be attached because it is not possible to invoke a method on a non existing target. Definition 35 captures these requirements.

Definition 35 (Agent creation). *A new agent is handled by the same processor as its target. An open target agent must have an attached and non-separate type.*

The type of an agent must show that the agent is on the same processor as the target. Definition 36 redefines section 8.27.17 of the Eiffel ECMA standard [4].

Definition 36 (Agent expression type). *Consider an agent expression with target type $T_x = (!, p, X)$ and feature f. Let $i_1, \ldots, i_m$ be the open argument positions and let $T_1, \ldots, T_m$ be the types of f's formal arguments at positions $i_1, \ldots, i_m$ (taking T_{i_1} to be T_x if the target is open, e.g. $i_1 = 0$). The agent expression has the following type:*

- *The type is $(!, p, PROCEDURE[(!, \bullet, X), (!, \bullet, TUPLE[T_1, \ldots, T_m])])$ if f is a procedure.*
- *The type is $(!, p, FUNCTION[(!, \bullet, X), (!, \bullet, TUPLE[T_1, \ldots, T_m]), T_R])$ if f is a function of result type T_R other than $(!, \bullet, BOOLEAN)$.*
- *The type is $(!, p, PREDICATE[(!, \bullet, X), (!, \bullet, TUPLE[T_1, \ldots, T_m])])$ if f is a function of result type $(!, \bullet, BOOLEAN)$.*

Example 17 (Agents). Listing 1.16 shows a class representing book traders.

Listing 1.16. trader class

```
class TRADER feature
  option: separate PROCEDURE[SELLER, TUPLE[separate BUYER]]

  prepare_option (a_seller: separate SELLER; a_book: separate BOOK;
      a_valuer: separate VALUER)
    do
      option := agent a_seller.sell_book (a_book, ?, a_valuer)
    end
end
```

The feature *prepare_option* creates an option to sell a particular book through a particular seller using a particular valuer at a later time. Profit can be generated if a book has been bought at a low price through the estimate of one valuer and if the book can be sold later at a higher price through the estimate of another valuer. In this example the option is represented by an agent for the feature *sell_book* with the seller as the target. The book and the valuer are closed arguments. The buyer is left as an open argument. The open argument is indicated with the question mark. The type of the agent is the type of the attribute *option*. The agent has the same processor tag as the target.

13 Once routines

A once routine gets executed at most once in a specified context. In Eiffel, a once routine either has a once per object, a once per thread, or a once per system semantics. If the once routine is a once function then the result gets shared within the specified context. Sections 8.23.20, 8.23.21, and 8.23.22 of the Eiffel ECMA standard [4] describe this in more details. In SCOOP, processors replace the notion of threads. In this section we will refine the existing Eiffel rules. Instead of the original options we consider a once per system or a once per processor semantics.

The result of a once function with a separate result type is supposed to be shared by different processors. Otherwise it makes no sense to declare the result as separate. Therefore such a function must have the once per system semantics. Once functions with a non-separate result type on the other hand must have a once per processor semantics. Otherwise there would be one object for multiple non-separate once functions on multiple processors. Clearly, the object can only be non-separate with respect to one processor. For all other processors the object would be a traitor. Once procedures do not come with these complications as they do not have a result. We assign a once per processor semantics to once procedures to give each processor the chance to make a fresh call to the procedure.

Definition 37 (Once routines semantics). *A once routine either has a once per system or a once per processor semantics.*

- *Once functions with a separate result type have the once per system semantics.*
- *Once functions with a non-separate result type have the once per processor semantics.*
- *Once procedures have the once per processor semantics.*

Example 18 (Once functions). Listing 1.17 shows a class representing phone directories of a country.

Listing 1.17. phone directory class

```
class PHONE_DIRECTORY feature
  national_directory: separate BOOK
    once
      ...
    end

  local_directory: BOOK
    once
      ...
    end
end
```

The country is divided into several states. Each state has a set of phone numbers. In addition there are some national phone numbers that are valid in every state. The phone directory takes this into account with two once functions: *national_directory* is a book containing all the national numbers and *local_directory* is a book with the local numbers. We imagine that each state is handled by a different processor and that the phone directory is on yet another processor. The feature *national_directory* is a separate once function. It has a once per system semantics. This takes into account that there is one directory for the whole nation. The feature *local_directory* is a non-separate once function and thus it has a once per processor semantics. This reflects the fact that there is one local directory per state.

14 Limitations and future work

At the beginning of this article we emphasized SCOOP's simplicity in comparison to semaphore based concurrent programming models. There are some threats to the validity of this claim. Our claim is not supported by any systematic study. Furthermore there has been progress on other concurrent programming models that make it easier to write correct and reusable concurrent programs. In particular there exist powerful concurrency libraries that can be used by developers, e.g. the concurrency libraries of Java (see e.g. [5]). A full support of our claim requires a study that compares SCOOP to the state-of-the art of concurrent programming models.

We do not claim that SCOOP programs run faster than other concurrent programs. However, performance is a key objective in any concurrent program. Performance of SCOOP programs is negatively affected if a centralized scheduling algorithm is used. A decentralized scheduling algorithm solves this issue and makes the system scalable. Performance can also be negatively influenced if the program under execution applies locking too coarsely. The differentiation between read- and write locks could improve the situation together with other refinements of the model. One refinement concerns wait by necessity. The SCOOP model can be optimized by only waiting when the result of the query is about to be accessed. As long as the result is not being accessed, it does not need to be available. A profiler for SCOOP specific metrics could help to find bottlenecks in SCOOP programs.

Currently SCOOP does not solve the asynchronous exception problem. Consider a situation where the execution of a procedure on a separate object results in an exception. It is possible that the client processor left the context of the feature call. In such a case the client processor is no longer able to handle the exception. The problem is tackled by Arslan and Meyer [1] as well as Brooke and Paige [2]. Arslan and Meyer define the guilty processor as the one who called the feature that triggered the exception. In their approach the target processor is considered busy by non-guilty processors. Only the guilty processor can resolve the situation by handling the exception as soon as the guilty processor locks the request queue of the busy processor once again. Brooke and Paige propose another mechanism to handle asynchronous exceptions. Their approach includes the notion of failed or dead objects.

Deadlocks are still an open problem in SCOOP. The selective locking mechanism is a useful technique to reduce the potential for deadlocks. However, this is not a method for ensure absence of deadlocks. It is necessary to conduct a comprehensive study on how deadlocks can occur in SCOOP programs. Such a study would facilitate an approach to avoid deadlocks in SCOOP programs. One approach in this direction is presented by Ostroff et al. [13]. They describe a virtual machine for SCOOP. The goal is to use model-checking and theorem proving methods to check global temporal logic properties of SCOOP programs.

The operational semantics used by Ostroff et al. may be extended to cover more of SCOOP. A complete definition could serve as a precise description of the model. At the moment SCOOP's complexity and the intrinsic details are hidden behind informal descriptions. The formalization could be the basis for formal proofs of properties promised by the model as well as for formal proofs of SCOOP programs. Interesting properties of the model include the absence of data races and the soundness of the type system.

Even though SCOOP naturally embraces distribution right from its start there are still open issues to be solved. In particular, it is unclear how distributed scheduling or mapping of processors to resources should be devised and implemented. Furthermore there is a fixed association of one object to a particular processor. It unclear whether this processor must be constant over time. Object migration would be especially beneficial for distribution because the latency of separate feature calls becomes significant in distributed programs.

The execution of a concurrent program can be different from one execution to the other. Hence, some bugs only show in some scheduling scenarios. This makes testing of concurrent applications very cumbersome. By design, SCOOP already rules out a number of scheduling related bugs such as high-level and low-level data races. Other potential bugs remain. It would be interesting to extend a testing framework to make scheduling a factor in test cases. Along the same line, it would be interesting to develop a debugger for SCOOP programs.

It would be interesting to have a design tool where one can graphically specify the dynamic and the static view of a SCOOP program. The dynamic view includes processors, the objects, and the interactions. The dynamic view uses concepts introduced in the static view. The static view shows the classes and existing SCOOP components. The graphical design is linked to the SCOOP code. Hence the designer can produce a SCOOP program out of the diagrams.

As part of the ETH Verification Environment (EVE) there is an implementation of SCOOP in progress. The implementation is available on our project website `http://scoop.origo.ethz.ch`.

References

1. Volkan Arslan and Bertrand Meyer. Asynchronous exceptions in concurrent object-oriented programming. In *Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages Proceedings*, pages 62–70, 2006.
2. Phillip J. Brooke and Richard F. Paige. Exceptions in concurrent eiffel. *Journal of Object Technology*, 6(10), 2007.
3. Edward G. Coffman, Melanie J. Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.
4. ECMA. Ecma-367 eiffel: Analysis, design and programming language 2nd edition. Technical report, ECMA International, 2006.
5. Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice.* Addison-Wesley, 2006.
6. C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, pages 102–116, 1971.
7. Bertrand Meyer. Sequential and concurrent object-oriented programming. In *Technology of Object-Oriented Languages and Systems*, pages 17–28, 1990.
8. Bertrand Meyer. Applying design by contract. *IEEE Computer*, 25(10):40–51, 1992.
9. Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):5680, 1993.
10. Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.
11. Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming.* PhD thesis, Swiss Federal Institute of Technology Zurich, 2007.
12. Piotr Nienaltowski and Bertrand Meyer. Contracts for concurrency. In *First International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages*, pages 27–49, 2006.
13. Jonathan S. Ostroff, Faraz Ahmadi Torshizi, Hai Feng Huang, and Bernd Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing*, 21(4):319–346, 2008.