Diss. ETH No. 21822

Prototyping a Concurrency Model

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

BENJAMIN MORANDI

Master of Science ETH in Computer Science
born on July 7th, 1983
citizen of Rossa (GR), Switzerland

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. José Meseguer, co-examiner
Prof. Dr. Jayadev Misra, co-examiner

2014

Acknowledgments

I am grateful to all those who contributed to the completion of this thesis. It was an honor for me to work with my adviser, Bertrand Meyer, and learn from his expertise in designing programming languages. His encouragement to find practical and simple solutions has significantly influenced the results in this thesis. His experience in writing has helped me improve my own technical writing skills. I also thank José Meseguer and Jayadev Misra for reviewing this thesis and giving me insightful questions and comments. Their feedback greatly helped me in compiling the final version of this thesis.

During my stay at the Chair of Software Engineering, I had the privilege to work with many competent and inspiring colleagues. First and foremost, I am grateful to Sebastian Nanz for guiding me to grow as a researcher. Our interesting and enjoyable collaboration and his numerous detailed comments have substantially contributed to this thesis. I am also thankful to Scott West for our collaboration on various topics and his helpful comments on performance analysis. Furthermore, I am grateful to Stephan van Staden for his comments on locks, to Martin Nordio for his role in extending the formal specification with exceptions, to Andreas Leitner for pointing out the need for data sharing, to Alexander Kogtenkov for his help in developing passive processors, to Piotr Nienaltowski for many interesting SCOOP-related discussions, and to Hassan Gomaa for our collaboration. I am also thankful to Till Bay, Manuel Oriol, and Marco Piccioni for kindly welcoming me to the group. Other members of our group were also an important source of support for me, namely Volkan Arslan, Cristiano Calcagno, Georgiana Caltais, Ilinca Ciupa, Christian Estler, Carlo A. Furia, Claudia Günthart, Alexey Kolesnichenko, Đurica Nikolić, Michela Pedroni, Yu (Max) Pei, Nadia Polikarpova, Chris Poskitt, Andrey Rusakov, Mischael Schill, Bernd Schoeller, Wolfgang Schwedler, Jiwon Shin, Mei Tang, Marco Trudel, Julian Tschannen, and Yi (Jason) Wei. It was a pleasure to work with many bright and motivated students, namely Florian Besser, Matteo Cortonesi, Daniel Furrer, Patrick Huber, Damien Müllhaupt, Andrey Nikonov, Ganesh Ramanathan, Emanuele Rudel, Andrey Rusakov, Mischael Schill, and Martino Trosi. I would also like to show my gratitude to the Eiffel Software team, in particular Jocelyn Fiat, Ian King, Alexander Kogtenkov, and Emmanuel (Manu) Stapf, for their continuous help with EVE.

My wife, Jiwon Shin, has been my beloved life companion and a valuable source of advice. She has provided a loving environment that allowed me to tackle challenges and grow. Her countless comments have greatly improved the quality of this thesis. My parents, Bruno Morandi and Christine Morandi, as well as my brothers, Michael Morandi, Simeon Morandi, and Matteo Morandi have supported, encouraged, and motivated me continuously.

This thesis was supported by the European Unions Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA). I am grateful for their support.

Benjamin Morandi

Abstract

Many novel programming models for concurrency have been proposed in the wake of the multicore computing paradigm shift. These models aim to raise the level of abstraction for expressing concurrency and synchronization in a program, thereby helping programmers avoid programming errors. This goal, however, causes the semantics of the models to become ever more complex and increases the risk of design flaws. Such flaws can have costly consequences if they are discovered after compiler and runtime support has been developed. It is therefore beneficial to verify the models beforehand.

This thesis proposes to prototype concurrency models using executable formal specifications. The prototype is useful from the beginning to the end of a model development. Initially, developers can use the prototype to test and correct the core of the model. As the development continues, developers can expand the prototype iteratively. For each extension, they enhance the specification, test the extension against the rest of the model, and apply corrections where necessary. Once the development is completed, the prototype serves as a reference.

This thesis applies the prototyping method to SCOOP, an object-oriented concurrency model. It demonstrates how the method facilitates the process of finding and resolving flaws in SCOOP and two extensions. In particular, it applies the method to extend SCOOP with (1) an exception mechanism to handle exceptions resulting from asynchronous calls and (2) a mechanism for fast and safe data sharing, reducing execution time by several orders of magnitude on data-intensive parallel programs. This effort results in 16 clarifications across various aspects, all included in a comprehensive executable formal specification in Maude.

This thesis also presents new SCOOP-specific performance metrics and a technique to compute them from event traces. Having a verified concurrency model does not guarantee that programmers write efficient concurrent programs. It is hence necessary to provide performance analysis tools that assist programmers in their task. Since SCOOP differs considerably from established models, reusing existing performance metrics is not an option. Instead, the new metrics are specifically designed for SCOOP. A case study on optimizing a robot control software demonstrates the usefulness of these metrics.

As a result of this thesis, SCOOP has an executable formal specification for future SCOOP developers, new mechanisms for exception handling and data sharing, as well as SCOOP-specific performance metrics. Having demonstrated the benefits of the method on an extensive concurrency model, we believe that the method can also benefit other models - new or mature.

Zusammenfassung

Im Zuge der zunehmenden Verbreitung von Mehrkernprozessoren wurden viele Programmiermodelle für Nebenläufigkeit vorgeschlagen. Diese Modelle helfen Programmierern Fehler zu vermeiden, indem sie Abstraktionen einführen, um Nebenläufigkeit und Synchronisation auszudrücken. Diese Abstraktionen erhöhen aber auch die Komplexität der Modelle, was das Risiko von Designfehlern erhöht. Solche Fehler können teure Konsequenzen haben, falls sie erst nach der Entwicklung des Compilers und der Laufzeit gefunden werden. Es ist daher vorteilhaft, das Modell vorweg zu verifizieren.

Diese Dissertation schlägt vor, Nebenläufigkeitsmodelle mittels ausführbaren formalen Spezifikationen zu entwickeln. Die Spezifikation eines Modells dient als Prototyp und ist nützlich vom Anfang bis zum Ende der Entwicklung. Zu Beginn können Entwickler den Prototyp verwenden, um den Kern des Modells zu testen und zu korrigieren. Später können Entwickler den Prototyp iterativ weiterentwickeln. Für jede Erweiterung ergänzen sie die Spezifikation, testen die Erweiterung in der Gesamtheit und bringen gegebenenfalls Korrekturen an. Sobald die Entwicklung des Modells abgeschlossen ist, dient der Prototyp als Referenz.

Diese Dissertation wendet die Prototyping Methode auf ein objektorientiertes Nebenläufigkeitsmodell namens SCOOP an. Sie zeigt wie die Methode hilft, Fehler in SCOOP und zwei Erweiterungen zu finden und zu korrigieren. Im Besonderen erweitert sie SCOOP unter Einsatz der Methode mit (1) einem neuen Mechanismus zur Behandlung von Ausnahmen, die aus asynchronen Aufrufen resultieren, und (2) einem Mechanismus zur schnellen und sicheren Datenteilung, welcher die Ausführungszeit von Programmen mit vielen Datenzugriffen um mehrere Grössenordnungen reduziert. Dieser Aufwand mündet in 16 Klarstellungen von verschiedenen Aspekten, die alle in einer umfassenden ausführbaren formalen Spezifikation in Maude enthalten sind.

Diese Dissertation präsentiert zusätzlich SCOOP-spezifische Leistungsmetriken und eine Technik, um diese aus Ereignisprotokollen zu berechnen. Ein verifiziertes Nebenläufigkeitsmodell garantiert nicht, dass Programmierer effiziente nebenläufige Programme schreiben. Es ist daher wichtig den Programmierern Leistungsanalysetools zur Verfügung zu stellen, die sie bei ihrer Arbeit unterstützen. Da sich SCOOP grundlegend von etablierten Modellen unterscheidet, ist es nicht angebracht, existierende Leistungsmetriken zu verwenden. Deshalb sind die neuen Metriken gezielt an SCOOP angepasst. Eine Fallstudie über die Optimierung einer Robotersteuerung demonstriert den Nutzwert dieser Metriken.

Durch diese Dissertation erhält SCOOP eine ausführbare formale Spezifikation für zukünftige SCOOP Entwickler, neue Mechanismen für die Ausnahmebehandlung und Datenteilung, sowie SCOOP-spezifische Leistungsmetriken. Da die Methode bei der Anwendung auf ein umfangreiches Nebenläufigkeitsmodell vorteilhaft ist, sind wir zuversichtlich, dass die Prototyping Methode auch bei der Entwicklung von anderen Modellen, neu oder ausgereift, von Vorteil sein kann.

Contents

1 Introduction

Almost all modern computer systems, ranging from servers to smartphones, have multiple *processing units* such as CPU cores, GPU cores, and co-processors. These processing units execute instructions in *parallel*, thus accelerating the system. To benefit from multiple units, a program must create multiple *threads*, each with a sequence of instructions that can be executed in parallel. The number of threads does not have to match the number of units; one unit can take several threads by *interleaving* their instructions. A system is *concurrent* when it executes a program with multiple threads in parallel or interleaved.

To develop such a *concurrent program*, the programming language in use must offer constructs to express threads and synchronization among each other. These constructs are defined by a *concurrency model*. Concurrency models can be divided into two main groups: models based on shared memory and models based on message passing. In *shared-memory*-based models, *client* threads spawns *supplier* threads with some workload. The threads read from and write to the same memory. In *message-passing* models, each thread can also spawn other threads, but each thread encapsulates its own private data. To access data of another thread, it must send a message with a request. The sender is called the *client*, and the receiver is called the *supplier*. The act of sending a message is either *synchronous*, i.e., the client waits until the supplier processed the message, or *asynchronous*, i.e., the client continues while the supplier processes the message.

A popular concurrency model based on shared memory is threads with monitors [88, 90], used in mainstream languages such as Java [82, 132] and C# [104]. In this model, threads use *monitors* to synchronize with each other before accessing the shared memory. Only one thread can lock the monitor at a given time; other threads must wait in the monitor's lock queue. The thread with the lock can wait on one of the monitor's *condition variables*, in which case it releases the lock and places itself in the queue for the condition variable. One of the next lock holders can then send a signal over the condition variable, moving all threads in the queue of the condition variable into the monitor's lock queue. Other concurrency models based on shared memory include the Parallel Random Access Machine (PRAM) [72, 81], the Partitioned Global Address Space (PGAS) model [6] (used in X10 [51], Fortress [4], and Chapel [118]), the tuple space model from Linda [79], the structured fork-join model used in Cilk [24, 102], and OpenMP's [172] underlying concurrency model.

An important message-passing model is the actor model [89], used in many languages including Erlang [66] and Scala [168]. The actor model represents threads as actors. An *actor* responds to incoming messages from other actors by creating new actors, sending messages to other actors, or determining new behavior for future messages. Other message-passing concurrency models include Communicating Sequential Processes (CSP) [91] used in occam [195], the π-calculus [156], the Orc process calculus [157] and its language [169], the Join Calculus [73] used in Cω [21], the Asynchronous Sequential Processes (ASP) calculus [49] used in ProActive [16], the Bulk Synchronous Parallel (BSP) model [208], Petri nets [177], the Reo coordination model [12], MPI's [146] underlying messaging model, and the rendezvous synchronization used in Ada [115].

Pure message-passing concurrency models avoid race conditions on data because each thread can only access its own data. The absence of data races eliminates a major source of concurrency errors, thus making it simpler to implement correct concurrent programs. This idea of simple development of concurrent programs is also at the heart of SCOOP [150, 164], an object-oriented concurrency model based on contracts. In SCOOP, processors encapsulate objects. A processor p can only operate on its own objects; to access an object of another processor q, it sends a *feature request* message to q by performing a *feature call*; in this context p is the client, and q is the supplier. To simplify development further, SCOOP provides constructs for automatic lock management, automatic conditional synchronization, and automatic waiting for query values. A client automatically waits until it has the locks of all suppliers and its precondition on the suppliers is satisfied before it calls features on the suppliers. If the client expects a result from a supplier, it automatically waits for the result.

SCOOP has gone through several iterations of refinement. In a series of articles [148–150], Meyer proposed SCOOP as a concurrency extension to Eiffel [105]. Nienaltowski [164] worked on usability, consistency, and compatibility with existing object-oriented constructs. To differentiate the two proposals, Meyer's work is called SCOOP_97, and Nienaltwoski's work is called SCOOP_07. Since then the concurrency model has also been applied to other languages [205]. Compton and Walker [56] provided the first implementation of SCOOP_97 as an extension of the SmartEiffel compiler [138]. Later, Fuks et al. [76] presented a source-to-source compiler from SCOOP_97 to Eiffel. As part of his work on the model [164], Nienaltowski presented a source-to-source compiler for SCOOP_07. EiffelSoftware integrated SCOOP_07 into the EiffelStudio [62] compiler and runtime, which is also available as a research platform called EVE [67].

While the SCOOP constructs are simple to use, their semantics is sophisticated, raising the possibility of design flaws. Flaws in a concurrency model can have costly consequences if they are discovered after compiler and runtime support has been developed. When users experience inexplicable bugs, they can lose their trust in the model. In the worst case, fixing these bugs requires many lines of code across many files to be changed, significantly prolonging the development time. It is therefore beneficial to verify the model prior to developing compiler and runtime support. One possibility is theorem proving (e.g., [136, 200]). Theorem proving can, however, be very time-consuming or otherwise only applicable to a small subset of the model.

To develop a sophisticated concurrency model like SCOOP, this thesis proposes to prototype the model using an *executable* formal specification. The executable specification lies between a purely descriptive specification and an implementation. It specifies the model, and at the same time it can be executed to see how the model behaves. The executable specification can thus be compared to an aircraft model. The aircraft model specifies an aircraft, and at the same time its aerodynamics can be tested in a wind tunnel. Using the executable formal specification, the model can be verified dynamically: the designers use test programs to check whether the formal executions conform to expectation. This approach combines the rigor and pureness of a formal model with the simplicity of testing. Once the development is completed, the executable formal specification serves as an unambiguous reference, which provides a starting point for future model extensions as well as compiler and runtime support.

1.1 Prototyping method

The *prototyping method* involves the following sequence of steps:

1. Start with an informal description of the main characteristics of the model. This step enables the development and discussion of the core ideas early on. Knowing the core ideas aids selection of the right formalism in the next step. It also accommodates existing informal descriptions.

2. From the informal description, develop an executable formal specification that covers all aspects of the model. This step provides the first opportunity to discover flaws in the model and provide clarifications.

3. Use the executable formal specification to test the model. Provide test programs and use the specification to analyze their executions. Address the design flaws directly in the specification. This step typically requires several iterations.

4. Use the clarified executable formal specification to develop compiler and runtime support.

1.2 Thesis statement

This thesis states that the prototyping method is beneficial to the development of a sophisticated concurrency model. The prototyping method can be applied to continue the development of an existing concurrency model and add new extensions to the model. In particular, the comprehensive executable formal specification can be used to test different combinations of model aspects, find flaws, and clarify the model. No compiler or runtime support is necessary, and the model can be studied in a purer environment.

This thesis establishes its claim by applying the method to SCOOP_07 – from now on simply called SCOOP – starting with the informal description by Nienaltowski [164]. SCOOP is an excellent study subject because it has a rich semantics and because no comprehensive formal specification exists yet; previous specifications [17, 37, 173] only cover a subset of SCOOP. To show that the method is also beneficial for new developments, this thesis extends SCOOP with two new mechanisms: an asynchronous exception mechanism and a mechanism for data sharing between processors.

1.3 Contributions and main results

This thesis makes the following contributions:

- *Prototyping method for concurrency models.* The method uses a comprehensive executable formal specification to test the model with a test suite.

- *Executable formal specification.* The executable formal specification integrates all aspects of SCOOP as well as the mechanism for asynchronous exceptions and the mechanism for data sharing; thus, it is capable of executing full-fledged SCOOP programs. It models a program's state with a number of abstract data types. The abstract data types permit a modular specification of the state on an abstract level and support a wide range of implementations. The executable formal specification uses a structural operational semantics to describe a program's executions in terms of the program elements. The specification is implemented in Maude [53, 54], a framework to specify and execute rewrite systems. We chose Maude because of its expressiveness and performance.

- *Clarifications.* Applying the prototyping method resulted in 16 clarifications based on flaws discovered in SCOOP and the extensions. These clarifications are a result of the comprehensive approach because some of them can only be found when studying the interplay between multiple aspects.

- *Exception mechanism.* The new exception mechanism for SCOOP handles exceptions resulting from asynchronous calls, called *asynchronous exceptions*. The mechanism is derived from a classification of approaches with new classifiers for propagation locations and for exception polling points. The classification also discusses the strengths and weaknesses of the approaches. The informal description of the mechanism is presented in a new conceptual framework with concepts to describe exception mechanisms.

  With the proposed mechanism, the client only propagates an asynchronous exception when it synchronizes with the supplier. Upon synchronization, the client waits for the supplier to finish all previous asynchronous calls; hence it knows about any past failure, and thus no data race can occur. The failed supplier reports failures as long as the client keeps a lock on it. Once the client releases the lock, it can no longer rely on the postconditions of the called features because another processor can intervene. Since a failure represents a failed postcondition, the supplier's failure becomes irrelevant, and thus the supplier can forget its failure. For applications where this is not suitable, the mechanism offers a safe mode where the client automatically synchronizes with a supplier before unlocking it. This semantics ensures that no processor has to handle the supplier's failure in a context where the failure is irrelevant.

- *Data sharing mechanism.* The new data sharing mechanism allows a processor to become *passive*. A client calling a feature on an object of a passive processor does not send a feature request; instead, it operates directly on the object after obtaining the lock on the passive processor. For frequent and short accesses, as common on data, this procedure is faster because it avoids the feature request overhead. Processors can thus assign shared data to a set of passive processors and then individually lock a passive processor subset to safely operate on the data. A case study shows that passive processors reduce the execution time by several orders of magnitude on data-intensive parallel programs.

- *Performance metrics.* Having a tested concurrency model does not ensure that programmers write efficient concurrent programs. Effective use of SCOOP requires tools to help programmers analyze and improve programs. Since SCOOP's execution semantics differs considerably from established models, this thesis introduces new performance metrics, tailored to the semantics of SCOOP. The *synchronization time*, for example, is the time it takes to obtain all missing locks and to establish the precondition. A case study on optimizing a robotic control software demonstrates the usefulness of the metrics, where the aforementioned synchronization time relates directly to the smoothness of the robot's gait.

The importance of verifying formal semantics of programming models has been recognized (see Section 5.6); to the best of our knowledge, however, no previous work has targeted a full concurrency model. This thesis is the first to use a comprehensive executable formal specification to test and improve a concurrency model.

1.4 Organization

The organization of the thesis follows the steps of the prototyping method, focusing on the development, testing, and clarification of the executable formal specification. Chapter 2 to Chapter 5 apply the prototyping method to SCOOP. Chapter 6 and Chapter 7 apply the method to develop the exception mechanism and the data sharing mechanism. Chapter 8 presents the performance metrics and Chapter 9 concludes the thesis with future work.

- Chapter 2 presents the existing informal description of SCOOP, as presented by Nienaltowski [164] in 2007. It summarizes the main concepts and introduces the terminology used throughout this thesis. The informal description is the starting point in applying the prototyping method to SCOOP.

- Chapter 3 presents the formal description of SCOOP along with the clarifications found during the formalization effort.

- Chapter 4 develops the main techniques of mapping the formal specification to Maude, namely mapping ADTs and transition rules. It also discusses how to use Maude's execution functionality to execute a program. These techniques are applied to the formal specification in Chapter 3 as well as the formal specifications for the asynchronous exception mechanism and the data sharing mechanism.

- Chapter 5 presents the test suite along with the flaws and clarifications found while testing SCOOP. It presents each flaw with a test program and the Maude input required to reproduce the issue. It clarifies the model by improving the formal specification.
- Chapter 6 applies the prototyping method to the development of the asynchronous exception mechanism. Following the steps of the method, this chapter first presents an informal description derived from a classification of existing approaches. It introduces a new conceptual framework to present the informal description. Based on the informal description, it extends the formal specification from Chapter 3. It then presents the results from the testing effort.
- Chapter 7 applies the prototyping method once again to develop the mechanism for data sharing. It first presents passive processors informally. It then extends the formal specification from Chapter 3 with passive processors and presents the testing results. Since performance cannot be evaluated with the executable formal specification alone, it uses a SCOOP implementation [67] with support for passive processors to experimentally demonstrate the performance benefits on data-intensive parallel programs.
- Chapter 8 introduces the performance metrics for SCOOP as well as a technique and a tool to compute the metrics from even traces. It evaluates the metrics with a case study on robotics control software.
- Chapter 9 summarizes the work of this thesis and concludes the thesis with future work.

This thesis discusses related work throughout the chapters to ensure proximity of relevant information among the wide range of topics. The related work discussions are distributed as follows: Chapter 5 discusses related work on developing and verifying formal semantics. Chapter 6 discusses related work on asynchronous exceptions. Chapter 7 discusses related work on data sharing in message-passing concurrency models. Chapter 8 discusses related work on performance analysis of concurrent programs.

The structure of this thesis guides the reader through our journey of developing SCOOP as it follows the steps of the prototyping method. Readers who are mainly interested in the final specification can consult Appendix A, which aggregates all clarifications and all extensions.

2 Informal description

SCOOP stands for *Simple Concurrent Object-Oriented Programming*. This name captures the main goal: to make the development of concurrent programs as simple as possible. Several factors contribute towards this goal. SCOOP eliminates data races, a major source of concurrency errors, thus making it simpler to implement concurrent programs correctly and reason about them. SCOOP provides constructs for automatic lock management, automatic conditional synchronization, and automatic waiting for query values, thus reducing programmers' responsibilities. SCOOP generalizes existing object-oriented concepts instead of introducing new ones, thus offering a familiar ground for programmers. This section introduces SCOOP informally as presented by Nienaltowski [164].

2.1 SCOOP in an example

The central idea of SCOOP is that every object is associated with a *processor*, called its *handler*. While an object can have only one handler, a processor can handle multiple objects. A processor can be a CPU, but it can also be implemented in software, for example, as a process or as a thread; any mechanism that can execute instructions sequentially is suitable. To access an object o, a processor p performs a *feature call* with *target* o. If o is handled by p, then p *applies* the called feature on o right away. If o is handled by a different processor q, then p sends a *feature request* to q. Processor q applies the feature on o, once it processed all previous feature requests; meanwhile, p can continue. In such a feature call, the caller p is the *client*; the handler of the target o is the *supplier*.

The producer-consumer problem serves as a simple illustration of these ideas. A root class defines the entities *producer*, *consumer*, and *buffer*. The keyword **separate** specifies that the referenced objects may be handled by a processor different from the current one.

```
producer: separate PRODUCER
consumer: separate CONSUMER
buffer: separate BUFFER
```

Figure 2.1 shows the objects in a processor diagram. A *processor diagram* is an extension of an UML object diagram [109] with curves that group objects according to their handler. Three processors p, g, and q handle the producer, the consumer, and the buffer with its content. Both the producer and the consumer access an unbounded buffer in feature calls such as *buffer.put* (n) and *buffer.item*. To access the buffer, they send requests instead of operating on the buffer directly.

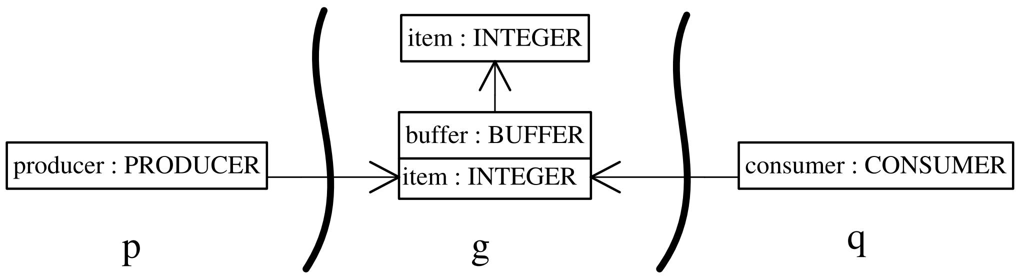


Figure 2.1: Producer-consumer processor diagram

To ensure exclusive access, the consumer and the producer lock the buffer before calling features on it. Such locking requirements of a feature must be expressed in the formal argument list: any target of separate type within the feature must occur as a formal argument; the arguments' handlers are locked for the duration of the feature execution, thus preventing data races. Such targets are called *controlled*. For instance, in *consume*, *buffer* is a formal argument; the consumer has exclusive access to the buffer while executing *consume*.

```
consume (buffer: separate BUFFER)
    -- Consume an item from the buffer.
  require
    not buffer.is_empty
  local
    consumed_item: INTEGER
  do
    consumed_item := buffer.item
  end

produce (buffer: separate BUFFER)
    -- Produce an item and put it into the buffer.
```

```
do
  buffer.put (data)
end
```

Condition synchronization relies on preconditions (**require** keyword) to express *wait conditions*. A precondition makes the execution of the feature wait until the condition is true. For example, the precondition of *consume* delays the execution until the buffer is not empty. As the buffer is unbounded, the corresponding producer feature does not need a precondition.

The runtime ensures that the result of the call *buffer.item* is properly assigned to the entity *consumed_item* using a mechanism called *wait by necessity* [47, 49]: while the consumer usually does not have to wait for an asynchronous call to finish, it will do so if it needs the result.

Automatic synchronization (based on wait by necessity, implicit locking, and wait conditions), absence of data races, and a single keyword to introduce concurrency – all these factors make SCOOP a concurrency model that is simple to use. Indeed, a recent empirical study by Nanz et al. [161] supports this claim. The runtime, however, is not simple because it takes responsibility for tasks that programmers no longer need to worry about. The following sections describe SCOOP and its runtime in more detail.

2.2 Programs

Figure 2.2 shows the EBNF [107] grammar for SCOOP programs and classes. For brevity, the grammar omits white space production rules, the concatenation symbol, and terminal symbol markings; it shows terminal symbols in color. For example, [”*expanded*”, *white_space*], ”*class*”, *white_space*, *class_name* becomes [**expanded**] **class** *class_name*. Furthermore, the grammar does not show production rules for comments such as −−comment.

A program is a set of classes with a *root class* and a *root procedure*. A class is either an *expanded class* (**expanded** keyword) or a *reference class* (no **expanded** keyword). Instances of an expanded class are copied whenever they are passed around; objects of a non-expanded class are aliased. As the focus of this thesis is on operational aspects rather than typing aspects, the thesis assumes that the class has no generic parameters and no inheritance declarations.

The class defines a set of *feature clauses* (**feature** keyword) each with a set of features. A *feature* is either a *routine*, i.e., a sequence of instructions, or an *attribute*, i.e., a data storage. If a routine returns a result, then it is a *function*; otherwise, it is a *procedure*. Functions and attributes are also called *queries*; routines are also called *commands*. After the **feature** keyword, a feature clause can define

```
program = {class} settings ;
settings = { class_name } . feature_name ;
class =
  [expanded] class class_name
  [create feature_name {, feature_name}]
  {feature [{ class_name {, class_name} }]
    {feature}}
  [invariant
    expression
    {expression}]
  end ;

feature = routine | attribute ;
routine =
  feature_name [alias " alias_name "]
      [( entity {; entity} )] [: type]
    [require
      expression
      {expression}]
    [local
      {entity}]
    (do | once)
      {instruction [;]}
    [ensure
      expression
      {expression}]
    [rescue
      {instruction [;]}]
    end ;
attribute = entity ;
entity = entity_name : type ;

class_name = name ;
feature_name = name ;
alias_name = not | + | - | * | / | // | \\ | ^ | .. | ⟨ | ⟩ | ⟨= | ⟩= | and | or | xor | and then | or
    else | implies ;
name = (a | . . . | z | A | . . . | Z) {a | . . . | z | A | . . . | Z};
entity_name = feature_name | Result | Current ;
```

Figure 2.2: SCOOP program grammar: classes and features

export restrictions, i.e., a list of classes whose objects can *call* the listed features on instances of the class. Without any restrictions, all objects can call the features on instances of the class; with a restriction {*NONE*}, only an instance of the class can call the features on itself. In the latter case, the features are *not exported*; in all other cases, the features are *exported*. The class declares a number of features as dedicated *creation procedures* (**create** keyword); these features can be used to create new objects. Finally, the class can define an invariant (**invariant** keyword) over its queries.

A routine has formal arguments and local variables (**local** keyword). Formal arguments are non-writable, and they must be different from attributes; however, when omitting the second restriction does not cause any confusion, this thesis overlooks the second restriction to improve readability. Next to explicit local variables, the routine has an implicit non-writable local variable **Current** for the object on which the routine is being executed. A function additionally has an implicit local variable **Result**; the function returns the value in **Result** when it returns.

Local variables, formal arguments, and attributes are known as *entities*. The Eiffel standard defines extensive initialization rules for entities; for simplicity, this thesis uses the *void* reference to initialize entities, i.e., initially entities do not reference any object.

A routine can have an alias (**alias** keyword), in which case the routine can additionally be used in infix notation. The routine can define a precondition (**require** keyword) and a postcondition (**ensure** keyword). It has a body with *instructions* (**do** or **once** keyword) to establish the postcondition given the precondition. If the routine is marked as a once routine (**once** keyword), then the body gets executed only once in a given context. Finally, the routine can define a rescue clause (**rescue** keyword) with instructions for exception handling.

Figure 2.3 shows the grammar for instructions and expressions. The *assignment instruction b* := *e* assigns the value of the expression *e* to the entity *b*. The *command instruction e0.f* (*e1*, ..., *en*) calls feature *f* on the *target expression e0* with *argument expressions e1*, ..., *en*. The target expression is optional; if not present, it is implicitly **Current**. The *creation instruction* **create** *b.f* (*e1*, ..., *en*) creates a new object in entity *b* by calling *f* with argument expressions *e1*, ..., *en*. The *if instruction* **if** *e* **then** *st* **else** *sf* **end** executes *st* if *e* is true and *sf* otherwise; the else part is optional. The simplified *loop instruction* **until** *e* **loop** *s* **end** executes *s* until *e* is true. The *retry instruction* **retry** is used for exception handling (see Section 6.4.1). An expression is either a literal, an entity, or a query expression; a query expression can also be in infix notation, provided the called feature has an alias. The **Void** literal represents the void reference.

```
instruction =
   entity_name := expression |
   [expression .] feature_name [( expression {, expression} )] |
   create entity_name . feature_name [( expression {, expression} )] |
   if expression then {instruction [;]} [else {instruction [;]}] end |
   until expression loop {instruction [;]} end |
   retry ;

expression =
   literal |
   entity_name |
   [expression .] feature_name [( expression {, expression} )] |
   [(] [expression] alias_name expression [)] ;
literal = boolean_literal | integer_literal | character_literal | void_literal ;
boolean_literal = True | False ;
integer_literal = [-](0 | . . . | 9) {0 | . . . | 9} ;
character_literal = ' (a | . . . | z | A | . . . | Z | 0 | . . . | 9) ' ;
void_literal = Void ;
```

Figure 2.3: SCOOP program grammar: instructions and expressions

2.3 Processors and objects

A processor p handles a set of objects; as the handler of these objects, p can operate on these objects directly. If p wants to call a feature on an object handled by a different processor q, then p needs to send a feature request to processor q. Processor q maintains a queue with requests resulting from feature calls on other processors; a lock protects access to the request queue. With the lock, p can add a feature request to this queue and processor q will execute the feature request as soon as it will have executed all previous feature requests in the request queue. Processor q uses its call stack to execute the feature request at the beginning of the request queue.

Definition 2.1 (Processor). A *processor p* is an autonomous thread of control capable of supporting the sequential execution of instructions on one or more objects. It consists of the following:

- *Handled objects*: objects associated to p
- *Request queue*: feature requests from other processors

- *Call stack*: stack for feature executions
- *Lock*: a lock protecting access to p
- *Locks*: locks held by p

Processor p can only operate on its handled objects; it is the *handler* of these objects. If p wants to call a feature on an object handled by a different processor q, then p sends a *feature request* to q; q then *applies the feature* as it goes through the following loop:

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.

□

Two objects handled by different processors are separate; two objects on the same processor are non-separate.

Definition 2.2 (Separateness of objects). 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. □

2.4 Types

A SCOOP type has three components to specify detachability, locality, and the class type. Figure 2.4 shows the grammar.

Definition 2.3 (Type). A type has a *detachable tag*, a *processor tag*, and a *class type*. The detachable tag specifies detachability, i.e., whether the typed entity can contain the void reference or not. If the detachable tag is *attached* (**attached** keyword), then the typed element cannot contain the void reference after initialization. If the detachable tag is *detachable* (**detachable** keyword), then the typed element can be void. In absence of both keywords, the typed element is attached.

The processor tag specifies locality. If the processor tag is *non-separate* (no **separate** keyword), then the typed element points to the handler of the current object. If the processor tag is *separate* (**separate** keyword), then the typed element

type =
  detachable_tag
  processor_tag
  class_type ;
detachable_tag = [**attached** | **detachable**] ;
processor_tag = [**separate** [*explicit_processor_specification*]] ;
explicit_processor_specification =
  qualified_explicit_processor_specification |
  unqualified_explicit_processor_specification ;
qualified_explicit_processor_specification = ⟨ *entity_name* **.handler** ⟩ ;
unqualified_explicit_processor_specification = ⟨ *entity_name* ⟩ ;
class_type = *class_name* ;

Figure 2.4: SCOOP program grammar: types

points to a potentially different processor; an element with an additional *explicit processor specification* points to the specified processor. An explicit processor specification is either qualified or unqualified. A *qualified explicit processor specification* is based on a non-writable, attached entity b; the specified processor is the handler of the object in b. An *unqualified explicit processor specification* is based on an attached processor attribute p of class type *PROCESSOR*; the specified processor is the processor stored in p. □

2.5 Feature applications

To apply a feature on behalf of a client q, the supplier p waits until it holds the locks of all processors $g \neq p$ handling an attached formal argument of reference type; a detachable formal argument indicates that p should not lock the argument's handler (*selective locking mechanism*), and a handler of an expanded argument must not be locked because p copies the object. Once p holds the lock on a processor, it can call features without interference from other processors. To ensure that all feature calls proceed without interference, the type system requires feature target expressions to be controlled.

Definition 2.4 (Controlled expression). An expression e is *controlled* if and only if e is attached, and e is non-separate or appears in an enclosing routine r that has an attached formal argument with the same handler as e. □

A non-separate target is handled by p itself, hence no interference can occur. A separate target with the same handler as an attached formal argument is exclusively available to p: for each formal argument, either p holds a lock, or the handler is p itself. To find a formal argument with the same handler, the type system checks whether the target expression e satisfies at least one of the following conditions:

- It appears as an attached formal argument of the enclosing routine r.
- It has a qualified explicit processor specification based on an entity w, and w is an attached formal argument of r.
- It has an unqualified explicit processor specification based on a processor attribute w, and some attached formal argument of r has w as its unqualified explicit processor specification.

Once p holds all the locks, it evaluates f's precondition. If f's precondition is violated and all its target expressions are controlled in the client q, then the precondition remains violated because q has exclusive access to all targets; hence p releases the locks and raises an exception in q. If f's precondition is not controlled, then p releases the locks and tries again later.

Once p holds all locks and the precondition is satisfied, p executes f. It then evaluates the postcondition. If any target in the postcondition is handled by p, then p asks each target handler to evaluate its part of the postcondition; p then combines the results. If no target is handled by p, then p asks any processor involved in the postcondition to lead the evaluation. After the postcondition, p evaluates the invariant if f is an exported routine (see Section 7.5 and Section 8.9.16 in [105]).

Finally, p releases the locks and returns. In case p returns an object o of expanded type to a client $q \neq p$, then q imports o: it creates a copied object structure where o and all non-separate objects reachable from o are replaced with copied objects handled by q. A simple copy of o is not sufficient, because the copy on q could have a non-separate entity, which would then reference an object on p. The following definition summarizes all these steps.

Definition 2.5 (Feature application). To *apply a feature f* on behalf of a client q, the supplier p takes the following steps:

1. *Synchronization.* Wait until all handlers $g \neq p$ of attached formal arguments of reference type are locked on behalf of p and the precondition of f holds. If the precondition is violated and controlled in q, then p raises an exception in q.
2. *Execution.* If f is a routine, then run its body. If f is an attribute, then evaluate it.

3. *Postcondition evaluation.* If at least one target in the postcondition is handled by p, ask each target handler to evaluate its part of the postcondition, then combine the results. If no target is handled by p, then ask any processor involved in the postcondition to lead the evaluation.

4. *Invariant evaluation.* If f is an exported routine, then evaluate the invariant.

5. *Lock releasing.* Ask each processor locked in the synchronization step to unlock after it is done with the feature requests issued by p.

6. *Result returning.* If f is a query, then return the result to q. If the result is of expanded type and $p = q$, then q must copy the result; if the result is of expanded type and $p \neq q$, then q must import the result.

□

2.6 Feature calls

To call a feature f on a supplier q, the client p first evaluates the target expressions and the argument expressions; p then passes the actual arguments to q. In case p passes an expanded object o to a supplier $q \neq p$, then q imports o to ensure that the copy of o on q does not have a non-separate entity with an object on p.

The client p then determines whether it needs to pass any locks. Lock passing eliminates a major source of deadlocks. Suppose the supplier q needs a lock on g to apply feature f. If p holds this lock, then q cannot proceed until p releases the lock. If f is a query then p waits for the result from q while holding the lock on g, thus preventing q to compute the result. A deadlock [55] has occurred. To avoid such deadlocks, p temporarily passes its locks if it determines that q needs any of them, i.e., if it attaches a controlled argument expression to an attached formal argument of reference type.

After passing locks, p generates a feature request for q. In case $p = q$, then p processes the feature request on its call stack. Otherwise it sends the feature request to q. In case the feature call is a *separate callback*, i.e., q has called p and now p calls back q, then q processes the feature request right away on its call stack while p waits. A deadlock could occur if p would add the feature request to the end of q's request queue: if both calls are synchronous, then p and q would wait for each other to terminate. To determine whether the feature call is a callback, p checks whether q held a lock on p. In case the call is not a separate callback, p adds the feature request to the end of q's request queue; p only waits, if it calls a query or if it needs to wait for passed locks to return. The following definition summarizes the process.

Definition 2.6 (Feature call). A client p performs the following steps to *call a feature f* with target expression e_0 and argument expressions $e_1, \ldots, e_n$:

1. *Target evaluation.* Evaluate the target expression e_0 with supplier q.
2. *Argument passing.* Evaluate the argument expressions $(e_1, \ldots, e_n)$ and bind them to the formal argument. If an actual argument is of expanded type and $p = q$, then q must copy the actual argument; if an actual argument is of expanded type and $p \neq q$, then q must import the actual argument.
3. *Lock passing.* Pass all locks to q if a controlled argument expression gets attached to an attached formal argument of reference type.
4. *Feature request.* Generate a feature request to apply f to the target.
   - If the feature call is non-separate, i.e., $p = q$, or the feature call is a separate callback, i.e., q held a lock on p, then ask q to process the feature request immediately using its call stack and wait for termination.
   - Otherwise, add the request to the end of q's request queue.
5. *Wait by necessity.* If f is a query, then wait for the result.
6. *Lock revocation.* If lock passing happened, then wait for the locks to come back.

□

The client and the supplier can be the same or they can be different. In the first case, the feature call is separate; in the second case, it is non-separate.

Definition 2.7 (Separateness of feature calls). A feature call is *separate* if the supplier is different from the client. Otherwise, the feature call is *non-separate*. □

2.7 Creation instructions

To create a new object on a processor q, a processor p executes a creation instruction on an entity b using a creation procedure f. The semantics of the creation instruction depends on the type of b. If b is of separate type without an explicit processor specification, then p creates a new processor with a new object. If b has an explicit specification and the specified processor does not exist yet, then p also creates a new processor with a new object; if the specified processor exists, p uses that processor for the new object. If b has a non-separate type, then p creates a new object on itself.

To ensure exclusive access, p makes sure to have q's lock if $q \neq p$; p then asks q to apply f (see Definition 2.6) and releases q's lock if necessary. The invariant of the new object must be satisfied after the application of f (see Section 7.5 and Section 8.9.16 in [105]). Processor q only evaluates the invariant if f is exported (see Definition 2.5). If f is not exported, then p should ask q to evaluate the invariant. To simplify the creation instruction, this thesis requires all creation procedures to be exported; thus p does not need to ask q. The following definition reflects this.

Definition 2.8 (Creation). A processor p performs the following steps to execute a creation instruction on entity b using a creation procedure f:

1. *Handler determination.* Determine the handler q of the new object.
   - If b is separate without an explicit processor specification, then create a new processor.
   - If b has an explicit processor specification and the specified processor already exists, then use the specified processor. If the specified processor does not exist yet, then create a new processor.
   - If b is non-separate, then use p.
2. *Locking.* Ensure $q \neq p$ is locked on behalf of p.
3. *Object creation.* Create a new object on q and attach it to b; ask q to execute the creation procedure f.
4. *Lock releasing.* If q has been locked in the locking step, then ask q to unlock after it is done with the creation.

□

2.8 Once routines

A once routine gets executed at most once in a context. If a once routine has been executed in the context, then it is called *not fresh* in the context. Otherwise it is called *fresh* in the context. The following definition defines the contexts.

Definition 2.9 (Once routine contexts). A once function f with a separate result type defined in a class type c has a once per system semantics: the objects of class type c in the system execute f at most once and then share f's result. A once function f with a non-separate result type defined in a class type c has a once per processor semantics: the objects of class type c handled by the same processor execute f at most once and then share f's result. A once procedure f defined in a class type c has a once per processor semantics: the objects of class type c handled by the same processor execute f at most once. □

Once routines complicate the import operation a bit. Consider a processor q importing an object o with a non-separate once function from a processor p. In case f is not fresh on q with result r and not fresh on p with a different result, then f must return the value on q when called on o's copy; there cannot be two different values for the same once function on the same processor. In case f is fresh on p, then f becomes not fresh on q with value r to preserve the existing result, as described in the following definition.

Definition 2.10 (Once routine import). The following table summarizes the effect of the import operation when a processor q imports an object with a non-separate once function f from a processor p:

	f is fresh on q.	f is not fresh on q.
f is fresh on p.	f remains fresh.	f remains not fresh (value of q).
f is not fresh on p.	f becomes not fresh (value of p).	f remains not fresh (value of q).

□

3 Formal specification

The informal description of SCOOP in the previous chapter provides an overview of SCOOP. It eases the understanding of the model as a whole, but it is too vague to ensure correct implementation. This chapter defines SCOOP formally to enable detailed and precise analysis of the model. The formal specification consists of two parts. The state specification describes the structure and the functionality of a program's state. The execution formalization describes the steps the system can take in a given state with a given workload. Together, they describe the executions of a program. Both parts assume a program to be given in an enriched intermediate representation instead of a code for rapid lookup of program elements.

3.1 Intermediate representation of programs

The formal specification assumes that a program is given in a type-checked and enriched intermediate representation. An intermediate representation is a concept borrowed from compiler theory, and its main purpose is the rapid lookup of program elements for further processing. In the intermediate representation for SCOOP, code elements are replaced with instances of abstract data types [135] or instances of structurally defined types; Figure 3.1 summarizes these types. For example, instead of browsing through the program text to find a feature based on its name, the intermediate representation provides the feature directly as an instance of an abstract data type.

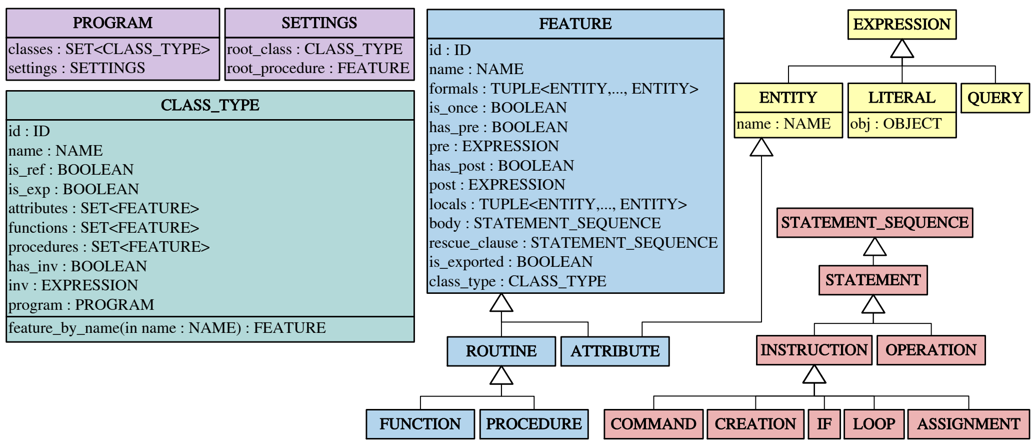


Figure 3.1: Types for the intermediate representation

3.1.1 Abstract data types

An abstract data type (ADT) [135] t is a type defined through its *features*, i.e., queries, commands, and constructors. A *query* of the form $name\colon t \rightarrow t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t_{n+1}$ takes the *target* of type t, i.e., the instance to be operated on, with further arguments of types $t_1, \ldots, t_n$ and returns information of type t_{n+1}. The specification uses the curried form (as in Haskell) instead of the equivalent Cartesian form $name\colon t \times t_1 \times \ldots \times t_n \rightarrow t_{n+1}$. The declaration of a *command* looks much like the one of a query; however, the result of a command is an *updated instance*, i.e., an updated version of the target. Calling a command on a target is hence called an *update of the target*. Different from a command or a query, a *constructor* does not take the target as an argument because its purpose is to create a new instance of type t.

An ADT feature can have a precondition, in which case the feature is partial, and the feature's signature has a crossed arrow $\nrightarrow$ instead of a normal arrow $\rightarrow$ after the type of each element that got restricted by the precondition. Axioms describe the effects of commands and constructors on queries. The declaration of a command or constructor lists all the axioms for the feature.

A generic ADT $t\langle G_1, \ldots, G_n\rangle$ with formal type parameters $G_1, \ldots, G_n$ is a type template rather than a single type. Using actual type parameters $t_1, \ldots, t_n$, the generic derivation $t\langle t_1, \ldots, t_n\rangle$ provides a type from the template.

For example, the following shows the generic ADT $STACK\langle G\rangle$ for a stack with elements of type G. The precondition (**require** keyword) of the *top* query states that the target s for *top* must not be empty. The axioms (**axioms** keyword) of the *push* command states that the updated stack has a new top element and that the stack is not empty. The constructor *make* returns an empty stack.

```
$empty\colon STACK\langle G\rangle \to BOOLEAN$
$top\colon STACK\langle G\rangle \nrightarrow G$
    $s.top$ **require**
        $\neg s.empty$
$push\colon STACK\langle G\rangle \to G \to STACK\langle G\rangle$
    **axioms**
        $s.push(e).top = e$
        $\neg s.push(e).empty$
$pop\colon STACK\langle G\rangle \nrightarrow STACK\langle G\rangle$
    $s.pop$ **require**
        $\neg s.empty$
    **axioms**
        $s.push(e).pop = s$
$make\colon STACK\langle G\rangle$
    **axioms**
        $make.empty$
```

To describe an ADT instance, one builds an expression starting with a constructor call. One then uses the resulting expression as the first argument in a command or query call, resulting in a new expression. One then repeats this process; the resulting nested expression describes the ADT instance. For better readability, this specification uses an equivalent object-oriented notation in which the first feature call is on the left and the last feature call is on the right. This notation does not show targets as arguments, but shows each target in front of the feature name, separated by a dot. This leads to the following translation between the functional and the object-oriented notation:

- The query expression $query(e_0, e_1, \ldots, e_n)$ in functional notation is equivalent to the object-oriented expression $e_0.query(e_1, \ldots, e_n)$.

- The command expression $command(e_0, e_1, \ldots, e_n)$ in functional notation is equivalent to the object-oriented expression $e_0.command(e_1, \ldots, e_n)$.

- The functional creation expression $constructor(e_1, \ldots, e_n)$ for an instance of an ADT t is equivalent to the expression **new** $t.constructor(e_1, \ldots, e_n)$ in object-oriented notation.

For $STACK\langle PROC\rangle$, a stack of processors, $empty(pop(push(make, p)))$ in functional notation is equivalent to **new** $STACK\langle PROC\rangle.make.push(p).pop.empty$ in object-oriented notation.

3.1.2 Basic abstract data types

A number of basic ADTs facilitate the types of code elements and the state specification. The generic ADT $SET\langle G\rangle$ specifies sets of elements with type G. The constructor *make* creates an empty set. The set can also be constructed intensionally as in $\{x \mid predicate(x)\}$ or extensionally as in $\{x_1, \ldots, x_n\}$. The command *add* takes an element and returns a set with the element. The query *has* returns whether an element is part of the set or not; the $\in$ notation is an alias for the *has* query. The query *empty* returns whether the set is empty or not, and the query *count* returns the number of elements in the set.

The generic ADT $TUPLE\langle G_1, \ldots, G_n\rangle$ has zero or more generic parameters for the types of the tuple elements; generic parameter G_i is the type of the element at position i. For example, the instances of $TUPLE\langle G\rangle$ are the tuples with one element of type G, and the instances of $TUPLE\langle G_1, G_2\rangle$ are the tuples with two elements of type G_1 and G_2 respectively. The type $TUPLE\langle G, \ldots, G\rangle$ includes tuples with any number of elements of type G, i.e., lists with elements of type G. The constructor *make* creates a tuple. The tuple expression $(x_1, \ldots, x_n)$ can also construct a tuple. The command *add* returns a tuple with a new element at the end. The query *has* checks whether an element is part of the tuple, and the query *empty* checks whether the tuple is empty. The query *count* returns the number of elements in the tuple, and the notation $(x_1, \ldots, x_n)(i)$ returns x_i.

The generic ADT $STACK\langle G\rangle$ specifies stacks with elements of type G. The constructor *make* creates a stack. The command *push* returns a stack with a new element at the top; the query *top* returns the top element, and *pop* removes the top element. The query *empty* returns whether the stack is empty or not, and the query *count* returns the number of elements in the stack. Finally, the query *flat* returns a set of type $SET\langle G\rangle$, merging all stack elements.

The ADT $MAP\langle K, G\rangle$ specifies maps with keys of type K and values of type G. The constructor *make* creates a map. The command *add* returns a map with a new mapping; the first argument is the key, and the second argument is the value. The query *keys* returns the set of all possible keys, and *val* returns the value for a given key.

Lastly, the ADT *BOOLEAN* has the two boolean values *true* and *false* as instances and offers the typical boolean operators. The ADT *NAME* specifies names and *ID* specifies identifiers.

3.1.3 Programs and settings

A program is an instance of *PROGRAM*. The program includes a set of classes (*classes* query), each of type *CLASS_TYPE*, as well as settings (*settings* query) with a root class (*root_class* query) and a root procedure (*root_procedure* query).

3.1.4 Class types

A class type is an instance of *CLASS_TYPE*. It has an identifier (*id* query) and a name (*name* query), and it is either a reference class type (*is_ref* query) or an expanded class type (*is_exp* query). Its features of type *FEATURE* are either attributes (*attributes* query), functions (*functions* query), or procedures (*procedures* query); the query *feature_by_name* returns a feature based on its name. The class type optionally defines an invariant (*has_inv* and *inv* queries). The invariant grammar in Figure 2.2 supports multiple expressions. The class type represents these expressions as a single conjunction. Lastly, the class type has a link (*program* query) to its program.

The class type *boolean* for boolean values in SCOOP programs (not to be confused with the *BOOLEAN* ADT for mathematical boolean values) is an instance of *CLASS_TYPE*. *boolean* is expanded, and it has an attribute *item* storing the mathematical boolean value, i.e., an instance of *BOOLEAN*.

3.1.5 Features

A feature is an instance of *FEATURE*. If the feature is an attribute, it is also an instance of *ATTRIBUTE*; a function is also an instance of *FUNCTION*, and a procedure is also an instance of *PROCEDURE*. A feature has an identifier (*id* query), a name (*name* query), and formal arguments (*formals* query) of type *ENTITY*. The feature can be exported (*is_exported* query), and it can be a once routine (*is_once* query). Further, it can have a precondition (*pre* and *has_pre* queries) and a postcondition (*post* and *has_post* queries), each represented as a single conjunction. It has local variables (*locals* query) of type *ENTITY* as well as a body (*body* query) and a rescue clause (*rescue_clause* query). Lastly, the feature has a link to the class it belongs to (*class_type* query); this link is useful to retrieve the invariant.

3.1.6 Statements

A *statement* of type *STATEMENT* is either an *instruction* of type *INSTRUCTION* or an *operation* of type *OPERATION*. An instruction occurs in code supplied by programmers whereas an operation is a run-time step. Figure 3.2 structurally defines *STATEMENT*; it also defines *STATEMENT_SEQUENCE* – i.e., sequences of statements – to type feature bodies and rescue clauses.

The notation is EBNF-like: types are structurally defined in terms of other types, EBNF-notation, and terminal symbols. For example, a command instruction $e_0.f(e_1, \dots, e_n)$ is of type *COMMAND*, and it consists of a target expression of type *EXPRESSION*, a dot, a feature of type *FEATURE*, as well as an argument list of type $TUPLE\langle EXPRESSION, \dots, EXPRESSION\rangle$.

```
STATEMENT_SEQUENCE ≙ [STATEMENT {; STATEMENT}] ;
STATEMENT ≙ INSTRUCTION | OPERATION ;
INSTRUCTION ≙ COMMAND | CREATION | IF | LOOP | ASSIGNMENT ;
COMMAND ≙
   EXPRESSION . FEATURE TUPLE⟨EXPRESSION, . . . , EXPRESSION⟩ ;
CREATION ≙
   create ENTITY . FEATURE TUPLE⟨EXPRESSION, . . . , EXPRESSION⟩ ;
IF ≙
   if EXPRESSION then
      STATEMENT_SEQUENCE
   [else
      STATEMENT_SEQUENCE]
   end ;
LOOP ≙ until EXPRESSION loop STATEMENT_SEQUENCE end ;
ASSIGNMENT ≙ ENTITY := EXPRESSION ;
RETRY ≙ retry ;
```

Figure 3.2: *STATEMENT*, *STATEMENT_SEQUENCE*, and *INSTRUCTION*

3.1.7 Expressions

An expression is an instance of *EXPRESSION*. An entity is an instance of *ENTITY* with name *name*. Two *ENTITY* instances are particularly important for features: *current*, i.e., the entity for the current object, and *result*, i.e., the entity for the result of a function.

Figure 3.3 defines literals and query expressions structurally. While the expression grammar in Figure 2.3 supports query expressions in prefix and infix notation; the intermediate representation represents all query expressions in prefix notation. To convert literals into objects, the universal function *obj* takes a literal and returns a new object matching the literal in both type and value.

3.1.8 Types

A type t is a triple (d, p, c): d is the detachable tag, p is the processor tag, and c is the class type of type *CLASS_TYPE*. The detachable tag d specifies detachability:

- If $d = !$ (**attached** keyword), then the typed element is statically guaranteed to hold a value, i.e., to be non-void.
- If $d = ?$ (**detachable** keyword), then the typed element can be void.

$\mathit{EXPRESSION} \triangleq \mathit{ENTITY} \mid \mathit{LITERAL} \mid \mathit{QUERY}$;
$\mathit{LITERAL} = \mathit{BOOL\_LITERAL} \mid \mathit{INT\_LITERAL} \mid \mathit{CHAR\_LITERAL} \mid \mathit{VOID\_LITERAL}$;
$\mathit{BOOL\_LITERAL} \triangleq$ **True** | **False** ;
$\mathit{INT\_LITERAL} \triangleq$ [**-**](**0** | . . . | **9**){**0** | . . . | **9**} ;
$\mathit{CHAR\_LITERAL} \triangleq$ **'** (**a** | . . . | **z** | **A** | . . . | **Z** | **0** | . . . | **9**) **'** ;
$\mathit{VOID\_LITERAL} \triangleq$ **Void** ;
$\mathit{QUERY} \triangleq$
$\quad \mathit{EXPRESSION}$ **.** $\mathit{FEATURE}\ \ \mathit{TUPLE}\langle \mathit{EXPRESSION}, \ldots, \mathit{EXPRESSION}\rangle$;

Figure 3.3: *EXPRESSION*

The processor tag p specifies locality:

- If $p = \top$ (**separate** keyword), then the typed element points to a potentially different processor.
- If $p =< x.handler >$ where x is of type *ENTITY* (**separate** keyword with qualified explicit processor specification), then the typed element points to the processor of the object in x.
- If $p =< x >$ where x is of type *ENTITY* (**separate** keyword with unqualified explicit processor specification), then the typed element points to the processor in x.
- If $p = \bullet$ (no keyword), then the typed element points to the current processor.

The *typing environment* Γ, formalized by Nienaltowski [164], contains the class hierarchy of a program along with all the type definitions for all features and entities. The predicate $\Gamma \vdash e : t$ denotes that expression e is of type t. The function $\mathit{type\_of}(\Gamma, e)$ returns the type of expression e in Γ. The predicate $\mathit{controlled}(\Gamma, t)$ denotes that expression e of type t is controlled. To establish $\mathit{controlled}(\Gamma, t)$ for e, one either finds an attached formal argument a in the enclosing routine with a type guaranteeing a and e are handled by the same processor, or one finds that e is non-separate. For the first case, a is the *controlling entity* for e; for the second case, the current entity is the controlling entity. The function $\mathit{controlling\_entity}(\Gamma, e)$ returns the controlling entity for an expression e as an instance of *ENTITY*. This function is essential to determine the handler of any controlled expression without evaluating the expression: one can simply determine the controlling entity and determine the handler of it.

3.2 State specification

Meyer [147] defines three levels on which a data structure, such as the state of a program, can be specified: functional, constructive, and physical. A functional specification is an algebraic characterization of the data structure. A constructive specification defines means to construct mathematical instances of the data structure. A physical specification describes the layout of instances in memory. A physical specification can be derived from a constructive one, and a constructive specification can be derived from a functional one. This section models the state with a number of ADTs, on the functional level in the above hierarchy. ADTs permit a modular specification of the state on an abstract level, supporting a wide range of implementations. The physical and the constructive specification can be derived from the ADTs.

3.2.1 Identifiers

Khoshafian and Copeland [121] describe several ways of identifying elements. An element's identity can be based on its value, it can be user-supplied, or it can be built-in, i.e., created and managed by the runtime. SCOOP objects and processors have a built-in identity. ADT instances do not have a built-in identity. Instead, their identity is defined by their value. To model a domain element with built-in identities as an ADT instance, the ADT needs an identifier query. Over time, a single domain element is then represented by a sequence of ADT instances with the same identifier. Every modification of the domain element is modeled as a new ADT instance where the value of the identifier query is preserved and all other queries modulo the modification are preserved. The formal specification ensures that no two ADT instances, modeling different domain elements, have the same identifier. To avoid duplicate identifiers, it assigns a fresh identifier for each ADT instance modeling a new domain element, using the universal stateful query *new_id* : *ID*. It then preserves the identifier.

3.2.2 Objects and references

Figure 3.4 shows the ADT for references. *REF* models references with an identity query *id* and a constructor *make*. The constructor uses the query *new_id* to create a fresh identifier for the newly created reference. The void reference *void* is an instance of this ADT.

Figure 3.5 shows the ADT for objects. Each object has a query *id* for its identifier, a query *class_type* for its class type, and a query *att_val* for its attribute values. An object can only have attribute values for attributes that are defined in its class type. The command *set_att_val* changes the value of an attribute f to v.

id: *REF* → *ID*
make: *REF*
  axioms
    make.id = new_id

Figure 3.4: *REF*

Only the attribute values for attributes defined in the class type can be modified. The value can either be a reference or a processor. Processor values are necessary to support processor attributes. The constructor *make* creates a new object with the given class type and initializes all the attribute values with the void reference. The query *copy* returns a copy of an object. The copied object has the same class type and the same attribute values as the original object, but it has a new identity.

3.2.3 Heap

The *heap* maps references to objects and manages once routines. Figure 3.6 shows the mapping-related features of the *HEAP* ADT. The query *objs* returns all objects, and the query *refs* returns all references to these objects. Here, *void* is not part of the reference set. The query *ref_obj* maps a reference to an object. The inverse query *ref* does the reverse. The query *last_added_obj* returns the reference to the last object added to the heap.

The command *add_obj* takes an object o, not yet on the heap, and adds it. The query *last_added_obj* returns the reference of the newly added object. The command *update_obj* takes a reference r and an updated object o and returns a heap where the reference r points to o. It requires that r is a valid reference and that o is an updated version of the original object.

Figure 3.7 shows the *HEAP* features concerned with once routines. For a processor p and a once routine f with identifier i, the query *fresh* returns whether f is fresh on p or not. For a once function f that is not fresh on a processor p, the query *once_result* returns the result of f on p. A once function f declared as separate (with or without an explicit processor specification) has a once per system semantics. Hence, the command *set_once_func_not_fresh* defines f as not fresh on all processors. A once function f with a non-separate result type has a once per processor semantics. Hence, *set_once_func_not_fresh* sets f as not fresh on p with the once result r. A once procedure f has a once per processor semantics. Consequently, *set_once_proc_not_fresh* sets f as not fresh on p. Finally, the command *set_once_rout_fresh* reverse the effect of the previous two.

```
$id\colon OBJ \to ID$
$class\_type\colon OBJ \to CLASS\_TYPE$
$att\_val\colon OBJ \to ATTRIBUTE \nrightarrow REF \cup PROC$
    $o.att\_val(f)$ **require**
        $o.class\_type.attributes.has(f)$
$set\_att\_val\colon OBJ \to ATTRIBUTE \nrightarrow REF \cup PROC \to OBJ$
    $o.set\_att\_val(f, v)$ **require**
        $o.class\_type.attributes.has(f)$
    **axioms**
        $o.set\_att\_val(f, v).att\_val(f) = v$
$make\colon CLASS\_TYPE \to OBJ$
    **axioms**
        $make(c).id = new\_id$
        $make(c).class\_type = c$
        $\forall i \in \{1, \ldots, n\}\colon make(c).att\_val(a_i) = void$
          **where**
            $\{a_1, \ldots, a_n\} \overset{def}{=} c.attributes$
$copy\colon OBJ \to OBJ$
    **axioms**
        $o.copy = make(o.class\_type)$
            $.set\_att\_val(a_1, o.att\_val(a_1))$
            $. \ldots$
            $.set\_att\_val(a_n, o.att\_val(a_n))$
          **where**
            $\{a_1, \ldots, a_n\} \overset{def}{=} o.class\_type.attributes$
```

Figure 3.5: *OBJ*

```
$objs\colon HEAP \to SET\langle OBJ\rangle$
$refs\colon HEAP \to SET\langle REF\rangle$
$ref\_obj\colon HEAP \to REF \nrightarrow OBJ$
    $h.ref\_obj(r)$ **require**
        $h.refs.has(r)$
$last\_added\_obj\colon HEAP \to REF$
    $h.last\_added\_obj$ **require**
        $\neg h.refs.empty$
$add\_obj\colon HEAP \to OBJ \nrightarrow HEAP$
    $h.add\_obj(o)$ **require**
        $\forall u \in h.objs\colon u.id \neq o.id$
        $\forall a \in o.class\_type.attributes\colon$
            $o.att\_val(a) \in REF \Rightarrow (o.att\_val(a) = void \lor h.refs.has(o.att\_val(a)))$
    **axioms**
        $h.add\_obj(o).objs = h.objs \cup \{o\}$
        $h.add\_obj(o).refs = h.refs \cup \{r\}$
        $h.add\_obj(o).ref\_obj(r) = o$
        $h.add\_obj(o).last\_added\_obj = r$
          **where**
            $r \overset{def}{=}$ **new** $REF.make$
$update\_obj\colon HEAP \to REF \nrightarrow OBJ \nrightarrow HEAP$
    $h.update\_obj(r, o)$ **require**
        $h.refs.has(r)$
        $o.id = h.ref\_obj(r).id$
        $\forall a \in o.class\_type.attributes\colon$
            $o.att\_val(a) \in REF \Rightarrow (o.att\_val(a) = void \lor h.refs.has(o.att\_val(a)))$
    **axioms**
        $h.update\_obj(r, o).objs.has(o)$
        $o \neq h.ref\_obj(r) \Rightarrow \neg h.update\_obj(r, o).objs.has(h.ref\_obj(r))$
        $h.update\_obj(r, o).ref\_obj(r) = o$
$ref\colon HEAP \to OBJ \nrightarrow REF$
    $h.ref(o)$ **require**
        $h.objs.has(o)$
    **axioms**
        $h.ref\_obj(h.ref(o)) = o$
```

Figure 3.6: *HEAP*: features to map references to objects

$\mathit{fresh}\colon \mathit{HEAP} \rightarrow \mathit{PROC} \rightarrow \mathit{ID} \nrightarrow \mathit{BOOLEAN}$

$\mathit{once\_result}\colon \mathit{HEAP} \rightarrow \mathit{PROC} \rightarrow \mathit{ID} \nrightarrow \mathit{REF}$

$\quad h.\mathit{once\_result}(p, i)$ **require**

$\qquad \neg h.\mathit{fresh}(p, i)$

$\mathit{set\_once\_func\_not\_fresh}\colon \mathit{HEAP} \rightarrow \mathit{PROC} \rightarrow \mathit{FEATURE} \nrightarrow \mathit{REF} \nrightarrow \mathit{HEAP}$

$\quad h.\mathit{set\_once\_func\_not\_fresh}(p, f, r)$ **require**

$\qquad f \in \mathit{FUNCTION} \wedge f.\mathit{is\_once}$

$\qquad r \neq \mathit{void} \Rightarrow h.\mathit{refs}.\mathit{has}(r)$

$\quad$ **axioms**

$\qquad (\exists d, c\colon \Gamma \vdash f : (d, \bullet, c)) \Rightarrow$

$\qquad\quad \neg h.\mathit{set\_once\_func\_not\_fresh}(p, f, r).\mathit{fresh}(p, f.\mathit{id}) \wedge$

$\qquad\quad h.\mathit{set\_once\_func\_not\_fresh}(p, f, r).\mathit{once\_result}(p, f.\mathit{id}) = r$

$\qquad (\exists d, c\colon \Gamma \vdash f : (d, p, c) \wedge p \neq \bullet) \Rightarrow \forall q \in \mathit{PROC}\colon$

$\qquad\quad \neg h.\mathit{set\_once\_func\_not\_fresh}(p, f, r).\mathit{fresh}(q, f.\mathit{id}) \wedge$

$\qquad\quad h.\mathit{set\_once\_func\_not\_fresh}(p, f, r).\mathit{once\_result}(q, f.\mathit{id}) = r$

$\mathit{set\_once\_proc\_not\_fresh}\colon \mathit{HEAP} \rightarrow \mathit{PROC} \rightarrow \mathit{FEATURE} \nrightarrow \mathit{HEAP}$

$\quad h.\mathit{set\_once\_proc\_not\_fresh}(p, f)$ **require**

$\qquad f \in \mathit{PROCEDURE} \wedge f.\mathit{is\_once}$

$\quad$ **axioms**

$\qquad \neg h.\mathit{set\_once\_proc\_not\_fresh}(p, f).\mathit{fresh}(p, f.\mathit{id})$

$\mathit{set\_once\_rout\_fresh}\colon \mathit{HEAP} \rightarrow \mathit{PROC} \rightarrow \mathit{FEATURE} \nrightarrow \mathit{HEAP}$

$\quad h.\mathit{set\_once\_rout\_fresh}(p, f)$ **require**

$\qquad f.\mathit{is\_once}$

$\quad$ **axioms**

$\qquad (f \in \mathit{FUNCTION} \wedge \exists d, c\colon \Gamma \vdash f : (d, \bullet, c)) \Rightarrow$

$\qquad\quad h.\mathit{set\_once\_rout\_fresh}(p, f).\mathit{fresh}(p, f.\mathit{id})$

$\qquad (f \in \mathit{FUNCTION} \wedge \exists d, c\colon \Gamma \vdash f : (d, p, c) \wedge p \neq \bullet) \Rightarrow \forall q \in \mathit{PROC}\colon$

$\qquad\quad h.\mathit{set\_once\_rout\_fresh}(p, f).\mathit{fresh}(q, f.\mathit{id})$

$\qquad (f \in \mathit{PROCEDURE}) \Rightarrow$

$\qquad\quad h.\mathit{set\_once\_rout\_fresh}(p, f).\mathit{fresh}(p, f.\mathit{id})$

Figure 3.7: *HEAP*: features for once routines

The command *set_once_func_not_fresh* handles once functions with any result type since the informal description does not impose any restrictions. This leads to an issue, as shown in the following classes:

```
class B create make feature
  p: PROCESSOR
  c: separate <p> C

  make
    do
      create c.make
    end

  f: separate <p> C
    once
      create Result.make
    end
end

class A create make feature
  make
    local
      b1, b2: B
    do
      create b1.make ; create b2.make
      g (b1)
    end

  g (b: B)
    local
      c: separate C
    do
      c := b.f
    end
end

class C create make feature
  make
    do end
end
```

The feature *make* in *A* creates two instances b_1 and b_2 of type *B*. After creation, both instances have a different value for p because each of them created an object on a new processor (q_1 in case of b_1; q_2 for b_2) and assigned that processor to p. The call to the once function f on b_1 causes b_1 to create another object on q_1. According to the informal description, such once functions have a once per system semantics. Hence, b_1 and b_2 share the result of f. For b_2, however, f returns a result that does not conform to the result type: the result is handled by q_1, but the type suggests that the result is handled by q_2. This issue would also occur if such once functions had a once per processor semantics. A similar example can be constructed for once functions with a qualified explicit processor specification. There are two solutions: either such once functions have a once per object semantics, or they must be disallowed. For simplicity, we propose a type system rule to invalidate them.

Clarification 3.1 (Once routines). Once routines must not have a result type with an explicit processor specification. □

Finally, the constructor *make* in Figure 3.8 creates a new heap with no objects and no references. All once routines are marked as fresh on all processors.

$make\colon HEAP$
  axioms
    $make.objs.empty$
    $make.refs.empty$
    $\forall p \in PROC, f \in FEATURE\colon\ f.is\_once \Rightarrow make.fresh(p, f.id)$

Figure 3.8: *HEAP*: constructors

3.2.4 Processors and regions

Figure 3.9 shows the ADT for processors. A processor has an identifier returned by the query *id*. The constructor *make* returns a new processor. The heap is partitioned [192] into disjoint *regions*, one region for each processor. The processor of a region is the handler for all objects in the region. The *REGIONS* ADT manages the partitions and the locks. Figure 3.10 shows the queries that map processors to objects. The query *procs* returns all processors in the system. The query *handled_objs* returns the references of a processor's handled objects, and *handler* does the inverse. Lastly, the query *last_added_proc* returns the last added processor.

```
$id\colon PROC \to ID$
$make\colon PROC$
    **axioms**
        $make.id = new\_id$
```

Figure 3.9: *PROC*

```
$procs\colon REGIONS \to SET\langle PROC\rangle$
$handled\_objs\colon REGIONS \to PROC \nrightarrow SET\langle REF\rangle$
    $k.handled\_objs(p)$ **require**
        $k.procs.has(p)$
$last\_added\_proc\colon REGIONS \nrightarrow PROC$
    $k.last\_added\_proc$ **require**
        $\neg k.procs.empty$
$handler\colon REGIONS \to REF \nrightarrow PROC$
    $k.handler(r)$ **require**
        $\exists p \in k.procs\colon k.handled\_objs(p).has(r)$
    **axioms**
        $k.handled\_objs(k.handler(o)).has(r)$
```

Figure 3.10: *REGIONS*: queries to map processors to objects

The informal description (see Chapter 2) associates one lock to each processor. This leads to an issue with separate callbacks. A separate callback is given if a processor p performs a feature call f to a processor q and q held a lock on p, as shown in Figure 3.11. To avoid a deadlock, p asks q to process the resulting feature request right away using the call stack. However, p does not hold a lock on q because this lock is held by processor g that locked q. To address this design flaw, the formal specification differentiates between two types of locks: request queue locks and call stack locks.

Clarification 3.2 (Locks). Each processor defines a *lock* for its *request queue* and a *lock* for its *call stack*. A lock on the request queue grants permission to add to the end of the request queue. A lock on the call stack grants permission to add to the top of the call stack. As long as a processor has not passed its locks, the processor is said to *hold* its locks. Independently of whether it has passed its locks, it is said to *claim* its locks. At creation, each processor holds its call stack lock.

For a separate callback, a processor q must pass its locks to a processor p and wait for the locks to return. Processor p can then use q's call stack lock to perform a separate callback to q. During the separate callback, processor p passes all its locks so that q can process the feature request. □

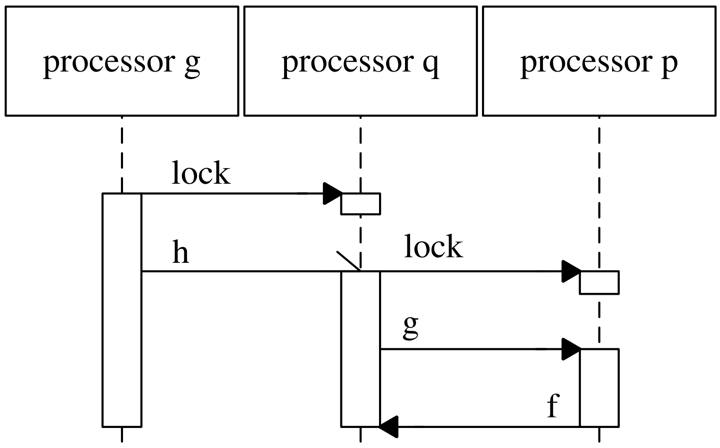


Figure 3.11: A separate callback. Processor g locks q and calls asynchronously. Processor q locks p and calls synchronously. Processor p calls back synchronously.

The queries shown in Figure 3.12 reflect this change. The feature *rq_locked* returns whether the request queue of a processor is locked or not; similarly, the feature *cs_locked* returns whether the call stack is locked. A number of queries manage the locks of a processor: the *obtained* locks, i.e., locks acquired by the processor, and the *retrieved* locks, i.e., locks retrieved from another processor during lock passing. The stack of sets return type models the way processors obtain locks. A processor goes through a nested series of feature applications and each feature application requires a set of locks. The query *obtained_cs_lock* always returns the obtained call stack lock of a processor. Lastly, the query *passed* returns whether a processor passed its locks.

```
$rq\_locked$: $REGIONS \rightarrow PROC \nrightarrow BOOLEAN$
    $k.rq\_locked(p)$ **require**
        $k.procs.has(p)$
$cs\_locked$: $REGIONS \rightarrow PROC \nrightarrow BOOLEAN$
    $k.cs\_locked(p)$ **require**
        $k.procs.has(p)$
$obtained\_rq\_locks$: $REGIONS \rightarrow PROC \nrightarrow STACK\langle SET\langle PROC\rangle\rangle$
    $k.obtained\_rq\_locks(p)$ **require**
        $k.procs.has(p)$
$obtained\_cs\_lock$: $REGIONS \rightarrow PROC \nrightarrow PROC$
    $k.obtained\_cs\_lock(p)$ **require**
        $k.procs.has(p)$
$retrieved\_rq\_locks$: $REGIONS \rightarrow PROC \nrightarrow STACK\langle SET\langle PROC\rangle\rangle$
    $k.retrieved\_rq\_locks(p)$ **require**
        $k.procs.has(p)$
$retrieved\_cs\_locks$: $REGIONS \rightarrow PROC \nrightarrow STACK\langle SET\langle PROC\rangle\rangle$
    $k.retrieved\_cs\_locks(p)$ **require**
        $k.procs.has(p)$
$passed$: $REGIONS \rightarrow PROC \nrightarrow BOOLEAN$
    $k.passed(p)$ **require**
        $k.procs.has(p)$
```

Figure 3.12: *REGIONS*: queries for locks

Figure 3.13 shows commands to add processors and objects. The command *add_proc* creates a new region for a processor p. Once added, p's request queue is unlocked and its call stack is locked. Apart from the initial lock on the call stack, p holds no locks and has not passed any locks. The command *add_obj* adds an object o referenced by r and associates it with a processor p. When o gets added, it must not yet be handled by any processor.

```
$add\_proc\colon REGIONS \to PROC \nrightarrow REGIONS$
  $k.add\_proc(p)$ **require**
    $\neg k.procs.has(p)$
  **axioms**
    $k.add\_proc(p).procs.has(p)$
    $k.add\_proc(p).last\_added\_proc = p$
    $k.add\_proc(p).handled\_objs(p).empty$
    $\neg k.add\_proc(p).rq\_locked(p)$
    $k.add\_proc(p).cs\_locked(p)$
    $k.add\_proc(p).obtained\_rq\_locks(p).empty$
    $k.add\_proc(p).obtained\_cs\_lock(p) = p$
    $k.add\_proc(p).retrieved\_rq\_locks(p).empty$
    $k.add\_proc(p).retrieved\_cs\_locks(p).empty$
    $\neg k.add\_proc(p).passed(p)$
$add\_obj\colon REGIONS \to PROC \nrightarrow REF \nrightarrow REGIONS$
  $k.add\_obj(p, r)$ **require**
    $k.procs.has(p)$
    $\forall q \in k.procs, x \in k.handled\_objs(q)\colon x.id \neq r.id$
  **axioms**
    $k.add\_obj(p, r).handled\_objs(p).has(r)$
```

Figure 3.13: *REGIONS*: commands to map processors to objects

A processor p uses the command *lock_rqs* from Figure 3.14 to lock the request queues of a set of processors $\overline{q}$. None of these request queues may be locked beforehand. At some point, processor p no longer requires the obtained request queue locks. It uses the command *pop_obtained_rq_locks* to remove its claims, provided it has not passed the locks. The request queues $\overline{q}$ remain locked until each of the locked processors finishes processing all feature requests and unlocks its request queues with a call to *unlock_rq*. In general, a request queue can only be unlocked once no other processors claims its lock.

```
$lock\_rqs\colon REGIONS \to PROC \nrightarrow SET\langle PROC\rangle \nrightarrow REGIONS$
  $k.lock\_rqs(p, \overline{l})$ **require**
    $k.procs.has(p)$
    $\forall x \in \overline{l}\colon k.procs.has(x)$
    $\forall x \in \overline{l}\colon \neg k.rq\_locked(x)$
  **axioms**
    $k.lock\_rqs(p, \overline{l}).obtained\_rq\_locks(p) = k.obtained\_rq\_locks(p).push(\overline{l})$
    $\forall x \in \overline{l}\colon k.lock\_rqs(p, \overline{l}).rq\_locked(x)$
$pop\_obtained\_rq\_locks\colon REGIONS \to PROC \nrightarrow REGIONS$
  $k.pop\_obtained\_rq\_locks(p)$ **require**
    $k.procs.has(p)$
    $\neg k.obtained\_rq\_locks(p).empty$
    $\neg k.passed(p)$
  **axioms**
    $k.pop\_obtained\_rq\_locks(p).obtained\_rq\_locks(p) = k.obtained\_rq\_locks(p).pop$
$unlock\_rq\colon REGIONS \to PROC \nrightarrow REGIONS$
  $k.unlock\_rq(p)$ **require**
    $k.procs.has(p)$
    $k.rq\_locked(p)$
    $\forall q \in k.procs\colon \neg k.obtained\_rq\_locks(q).flat.has(p)$
  **axioms**
    $\neg k.unlock\_rq(p).rq\_locked(p)$
```

Figure 3.14: *REGIONS*: commands for locking and unlocking

To pass its locks to a processor q, p calls the command *pass_locks* from Figure 3.15 with a set of request queue locks $\overline{l_r}$ and a set of call stack locks $\overline{l_c}$ to be passed. At this instance, p must hold all these locks. Because $\overline{l_r}$ and $\overline{l_c}$ can potentially be empty, the command only marks p's locks as passed if at least one of the two sets of locks is non-empty. If a processor q different from p passed its locks previously, and now p passes these locks back to q, the command marks q's locks as not passed. This case is important for handling separate callbacks.

$$
\begin{array}{l}
\textit{pass\_locks}\colon \textit{REGIONS} \to \textit{PROC} \nrightarrow \textit{PROC} \nrightarrow \textit{TUPLE}\langle \textit{SET}\langle \textit{PROC}\rangle, \textit{SET}\langle \textit{PROC}\rangle\rangle \\
\qquad\qquad \nrightarrow \textit{REGIONS} \\
\quad k.\textit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c}))\ \textbf{require} \\
\qquad k.\textit{procs.has}(p) \wedge k.\textit{procs.has}(q) \\
\qquad \forall x \in \overline{l_r}\colon k.\textit{obtained\_rq\_locks}(p).\textit{flat.has}(x) \vee k.\textit{retrieved\_rq\_locks}(p).\textit{flat.has}(x) \\
\qquad \forall x \in \overline{l_c}\colon x = k.\textit{obtained\_cs\_lock}(p) \vee k.\textit{retrieved\_cs\_locks}(p).\textit{flat.has}(x) \\
\qquad \neg\overline{l_r}.\textit{empty} \vee \neg\overline{l_c}.\textit{empty} \Rightarrow \neg k.\textit{passed}(p) \\
\quad \textbf{axioms} \\
\qquad \neg\overline{l_r}.\textit{empty} \vee \neg\overline{l_c}.\textit{empty} \Rightarrow k.\textit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\textit{passed}(p) \\
\qquad k.\textit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\textit{retrieved\_rq\_locks}(q) = k.\textit{retrieved\_rq\_locks}(q).\textit{push}(\overline{l_r}) \\
\qquad k.\textit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\textit{retrieved\_cs\_locks}(q) = k.\textit{retrieved\_cs\_locks}(q).\textit{push}(\overline{l_c}) \\
\qquad \left(\begin{array}{l}
p \neq q \wedge \\
k.\textit{passed}(q) \wedge \\
k.\textit{obtained\_rq\_locks}(q).\textit{flat} \subseteq \overline{l_r} \wedge \\
k.\textit{retrieved\_rq\_locks}(q).\textit{flat} \subseteq \overline{l_r} \wedge \\
k.\textit{obtained\_cs\_lock}(q) \in \overline{l_c} \wedge \\
k.\textit{retrieved\_cs\_locks}(q).\textit{flat} \subseteq \overline{l_c}
\end{array}\right) \Rightarrow \begin{array}{l}
\neg k.\textit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})) \\
\quad .\textit{passed}(q)
\end{array}
\end{array}
$$

Figure 3.15: *REGIONS*: command for passing locks

To revoke the passed locks from q, p uses the command *revoke_locks* from Figure 3.16. The previous lock passing operation from p to q might have marked the locks of q as not passed; revoking the locks requires the reverse action. If after removing p's locks from q's stack, p has retrieved locks from q, then q's locks must be marked as passed since they are now held by p.

```
$revoke\_locks \colon REGIONS \rightarrow PROC \nrightarrow PROC \nrightarrow REGIONS$
  $k.revoke\_locks(p, q)$ **require**
    $k.procs.has(p) \wedge k.procs.has(q)$
    $\neg k.retrieved\_rq\_locks(q).empty \wedge \neg k.retrieved\_cs\_locks(q).empty$
    $k.retrieved\_rq\_locks(q).top \subseteq$
        $k.obtained\_rq\_locks(p).flat \cup k.retrieved\_rq\_locks(p).flat$
    $k.retrieved\_cs\_locks(q).top \subseteq$
        $\{k.obtained\_cs\_lock(p)\} \cup k.retrieved\_cs\_locks(p).flat$
    $k.retrieved\_rq\_locks(q).top \cup k.retrieved\_cs\_locks(q).top \neq \{\} \Rightarrow k.passed(p)$
    $\neg k.passed(q)$
  **axioms**
    $k.retrieved\_rq\_locks(q).top \cup k.retrieved\_cs\_locks(q).top \neq \{\} \Rightarrow$
        $\neg k.revoke\_locks(p, q).passed(p)$
    $k.revoke\_locks(p, q).retrieved\_rq\_locks(q) = k.retrieved\_rq\_locks(q).pop$
    $k.revoke\_locks(p, q).retrieved\_cs\_locks(q) = k.retrieved\_cs\_locks(q).pop$
```

$$
\left(\begin{array}{l}
p \neq q \wedge \\
\left(\begin{array}{l}
\exists x \in k.retrieved\_rq\_locks(p).flat \colon (\\
\quad k.obtained\_rq\_locks(q).flat.has(x) \vee \\
\quad k.retrieved\_rq\_locks(q).pop.flat.has(x) \\
) \vee \\
\exists x \in k.retrieved\_cs\_locks(p).flat \colon (\\
\quad x = k.obtained\_cs\_lock(q) \vee \\
\quad k.retrieved\_cs\_locks(q).pop.flat.has(x) \\
)
\end{array}\right)
\end{array}\right)
\Rightarrow
\begin{array}{l}
k.revoke\_locks(p, q) \\
\quad .passed(q)
\end{array}
$$

Figure 3.16: *REGIONS*: command for revoking locks

The constructor *make* from Figure 3.17 creates a new instance of *REGIONS* with no processors.

$make\colon REGIONS$
  axioms
    $make.procs.empty$

Figure 3.17: *REGIONS*: constructors

3.2.5 Environments and store

For every feature a processor executes, it creates a new variable *environment* to store the values of formal arguments, local variables, the current object entity, and the result entity (only for functions). Figure 3.18 shows the *ENV* ADT with a query *names* to store the defined names and the query *val* to get the value for each name. The command *update* assigns a new value to a name. The constructor *make* returns an empty environment.

$names\colon ENV \to SET\langle NAME\rangle$
$val\colon ENV \to NAME \nrightarrow REF \cup PROC$
  $e.val(n)$ **require**
    $e.names.has(n)$
$update\colon ENV \to NAME \to REF \cup PROC \to ENV$
  axioms
    $e.update(n, v).names = e.names \cup \{n\}$
    $e.update(n, v).val(n) = v$
$make\colon ENV$
  axioms
    $make.names.empty$

Figure 3.18: *ENV*

The *store* contains the environment stacks of all processors. The ADT *STORE* in Figure 3.19 has a single query *envs* that returns a stack of environments for a processor. The command *push_env* pushes an environment on top a processor's stack, and the command *pop_env* pops the top environment from a non-empty stack of environments. The constructor *make* creates an empty store.

$envs\colon STORE \to PROC \to STACK\langle ENV\rangle$
$push\_env\colon STORE \to PROC \to ENV \to STORE$
  axioms
    $s.push\_env(p, e).envs(p) = s.envs(p).push(e)$
$pop\_env\colon STORE \to PROC \nrightarrow STORE$
  $s.pop\_env(p)$ **require**
    $\neg s.envs(p).empty$
  axioms
    $s.pop\_env(p).envs(p) = s.envs(p).pop$
$make\colon STORE$
  axioms
    $\forall p \in PROC\colon make.envs(p).empty$

Figure 3.19: *STORE*

3.2.6 State

The ADT *STATE* in Figure 3.20 has three queries that return the different components of the state. The command *set* sets the regions, the heap, and the store. A precondition specifies consistency criteria. The first two clauses require that a processor can handle an object if and only if the object is on the heap. The third clause requires that if the heap declares a feature as not fresh on a processor p, then the processors must have a region. The fourth clause requires that all processors stored in attribute values have a region. The fifth clause requires that each non-empty environment in the store must belong to a processor that has a region. The sixth precondition requires that each value in the store must either be a known reference or a known processor. To constructor *make* creates a new state with a new heap, a new store, and new regions.

$regions\colon STATE \to REGIONS$
$heap\colon STATE \to HEAP$
$store\colon STATE \to STORE$
$set\colon STATE \to REGIONS \nrightarrow HEAP \nrightarrow STORE \nrightarrow STATE$
  $\sigma.set(k, h, s)$ **require**
    $\forall p \in k.procs, r \in k.handled\_objs(p)\colon h.objs.has(h.ref\_obj(r))$
    $\forall o \in h.objs\colon \exists p \in k.procs\colon h.ref(o) \in k.handled\_objs(p)$
    $\forall p \in PROC, f \in FEATURE\colon \neg h.fresh(p, f.id) \Rightarrow k.procs.has(p)$
    $\forall o \in h.objs, a \in o.class\_type.attributes\colon$
      $o.att\_val(a) \in PROC \Rightarrow k.procs.has(o.att\_val(a))$
    $\forall p \in PROC, e \in s.envs(p)\colon \neg e.names.empty \Rightarrow k.procs.has(p)$
    $\forall p \in k.procs, e \in s.envs(p), x \in e.names\colon$
      $(e.val(x) \in REF \Rightarrow e.val(x) = void \lor h.refs.has(e.val(x)))\land$
      $(e.val(x) \in PROC \Rightarrow k.procs.has(e.val(x)))$
  axioms
    $\sigma.set(k, h, s).regions = k$
    $\sigma.set(k, h, s).heap = h$
    $\sigma.set(k, h, s).store = s$
$make\colon STATE$
  axioms
    $make.regions =$ **new** $REGIONS.make$
    $make.heap =$ **new** $HEAP.make$
    $make.store =$ **new** $STORE.make$

Figure 3.20: *STATE*: components features

STATE offers a *facade* with convenient access to the state functionality and further features that operate on all the components instead of just one of them. For example, the following expression defines a new state with a new processor: $\sigma' \overset{def}{=} \sigma.set(\sigma.regions.add\_proc(\textbf{new}\ PROC.make), \sigma.heap, \sigma.store)$. This expression is too long for this simple task, especially if the expression appears multiple times. The facade provides a simple convenience command that updates the state accordingly.

Mapping of processors to objects and mapping of references to objects

Figure 3.21 and Figure 3.22 show features for the mapping between processors, objects, and references. The query *new_proc* is a shorthand for processor creation, and the query *new_obj* is a shorthand for object creation.

$procs\colon STATE \rightarrow SET\langle PROC\rangle$
    axioms
        $\sigma.procs = \sigma.regions.procs$
$last\_added\_proc\colon STATE \nrightarrow PROC$
    $\sigma.last\_added\_proc$ **require**
        $\neg\sigma.regions.procs.empty$
    axioms
        $\sigma.last\_added\_proc = \sigma.regions.last\_added\_proc$
$handler\colon STATE \rightarrow REF \nrightarrow PROC$
    $\sigma.handler(r)$ **require**
        $\sigma.heap.refs.has(r)$
    axioms
        $\sigma.handler(r) = \sigma.regions.handler(r)$
$last\_added\_obj\colon STATE \nrightarrow REF$
    $\sigma.last\_added\_obj$ **require**
        $\neg\sigma.heap.refs.empty$
    axioms
        $\sigma.last\_added\_obj = \sigma.heap.last\_added\_obj$
$ref\_obj\colon STATE \rightarrow REF \nrightarrow OBJ$
    $\sigma.ref\_obj(r)$ **require**
        $\sigma.heap.refs.has(r)$
    axioms
        $\sigma.ref\_obj(r) = \sigma.heap.ref\_obj(r)$
$ref\colon STATE \rightarrow OBJ \nrightarrow REF$
    $\sigma.ref(o)$ **require**
        $\sigma.heap.objs.has(o)$
    axioms
        $\sigma.ref(o) = \sigma.heap.ref(o)$

Figure 3.21: *STATE*: facade features for the mapping between processors, objects, and references

```
$add\_proc : STATE \rightarrow PROC \nrightarrow STATE$
    $\sigma.add\_proc(p)$ **require**
        $\neg \sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.add\_proc(p) = \sigma.set(\sigma.regions.add\_proc(p), \sigma.heap, \sigma.store)$
$add\_obj : STATE \rightarrow PROC \nrightarrow OBJ \nrightarrow STATE$
    $\sigma.add\_obj(p, o)$ **require**
        $\sigma.regions.procs.has(p)$
        $\forall u \in \sigma.heap.objs : u.id \neq o.id$
        $\forall a \in o.class\_type.attributes :$
            $(o.att\_val(a) \in REF \Rightarrow o.att\_val(a) = void \vee \sigma.heap.refs.has(o.att\_val(a))) \wedge$
            $(o.att\_val(a) \in PROC \Rightarrow \sigma.regions.procs.has(o.att\_val(a)))$
    **axioms**
        $\sigma.add\_obj(p, o) = \sigma.set(k, h, s)$
          **where**
            $h \overset{def}{=} \sigma.heap.add\_obj(o)$
            $k \overset{def}{=} \sigma.regions.add\_obj(p, h.last\_added\_obj)$
            $s \overset{def}{=} \sigma.store$
$update\_obj : STATE \rightarrow REF \nrightarrow OBJ \nrightarrow STATE$
    $\sigma.update\_obj(r, o)$ **require**
        $\sigma.heap.refs.has(r)$
        $o.id = \sigma.heap.ref\_obj(r).id$
        $\forall a \in o.class\_type.attributes :$
            $(o.att\_val(a) \in REF \Rightarrow o.att\_val(a) = void \vee \sigma.heap.refs.has(o.att\_val(a))) \wedge$
            $(o.att\_val(a) \in PROC \Rightarrow \sigma.regions.procs.has(o.att\_val(a)))$
    **axioms**
        $\sigma.update\_obj(r, o) = \sigma.set(\sigma.regions, \sigma.heap.update\_obj(r, o), \sigma.store)$
$new\_proc : STATE \rightarrow PROC$
    **axioms**
        $\sigma.new\_proc =$ **new** $PROC.make$
$new\_obj : STATE \rightarrow CLASS\_TYPE \rightarrow OBJ$
    **axioms**
        $\sigma.new\_obj(c) =$ **new** $OBJ.make(c)$
```

Figure 3.22: *STATE*: facade features for the mapping between processors, objects, and references

Importing

Expanded objects must be imported when they cross a processor boundary. When a processor *p* passes an expanded object *o* to a processor *q*, the import operation copies the object structure such that *o* and all non-separate objects are replaced with copied objects handled by *q*. The import operation can produce an object structure that contains both copied and original objects. As noted by [164], this can be an issue in case one of the copied objects has an invariant based on the identities of objects. For example, Figure 3.23 shows two objects *a* and *b* handled by processor *p* and another object *c* handled by *g*. Object *a* has an invariant with a query $c.b = b$. The import operation on *a* executed by processor *q* results in two new objects a' and b' on *q*. The attribute *c* of object a' points to the old *c*, and *b* points to the new object b'. Object a' is inconsistent because $c.b$ and *b* identify different objects, namely *b* and b'.

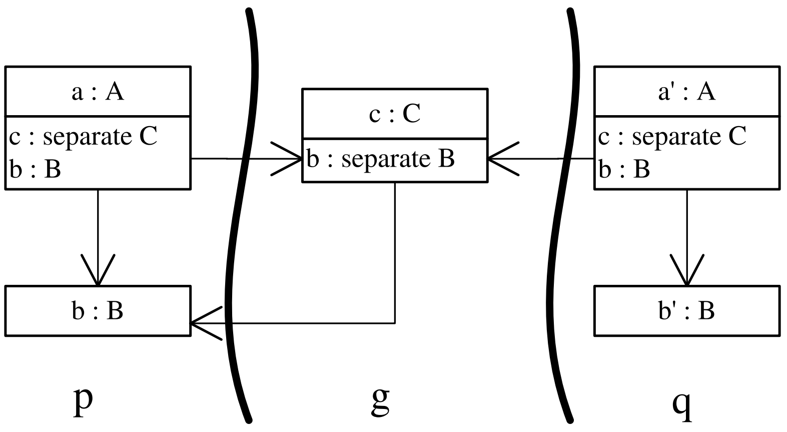


Figure 3.23: Invariant violation in an imported object structure

The deep import operation is a variant of the import operation that does not mix the copied and the original objects. Instead of copying only non-separate objects, it makes a full copy of the object structure. The importing processor handles all the copies of the non-separate objects, and the copy of a separate object is handled by the processor of the original object. Unfortunately, the deep import operation has critical drawbacks. First, many more objects must be copied. Second, preventing data races is extremely costly as the operation must atomically lock the handlers of all reachable separate objects before a consistent copy can be made. These drawbacks outweigh the benefits of the deep import operation; thus the formal specification uses the import operation, but introduces a convention: in imported object structures, invariants must not rely on object identities.

Clarification 3.3 (Import). The deep import operation atomically locks the handlers of all reachable separate objects, which is too costly. The import operation is a better fit, but requires a convention: in imported object structures, invariants must not rely on object identities. □

Once routines complicate the import operation. Consider a processor q about to import an object o handled by a different processor p, where o has a non-separate once function f. The informal description defines that in case f is not fresh on p and fresh on q, the import operation uses the value on p as the new value on q (see Figure 3.24). This causes an issue to arise. On q the once function returns a reference to an object on p; however, the result type of f indicates that the object should be handled by q. The problem can be solved either by the import operation also importing the value of f and using it as the value on q, or the import operation keeping f as fresh on q. The latter option is simpler and more efficient, but it can cause an invariant on the imported object structure to be violated, should the invariant rely on the value of f. Thus the formal specification introduces another convention: in imported object structures, non-separate once functions must return results that satisfy the object structure's invariants when recomputed on any processor. This convention also prevents violated invariants in case f is not fresh on both p and q. The following clarification captures the convention and generalizes the import operation revision for non-separate once functions and once procedures.

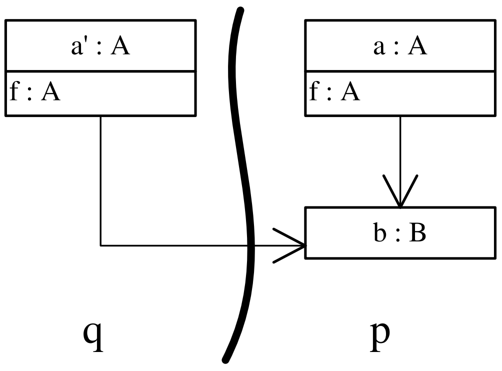


Figure 3.24: Type system unsoundness due to a once routine in an imported object

Clarification 3.4 (Import and once routines). Let q be a processor about to import an object o handled by a different processor p, where o has a non-separate once function or once procedure f. In case f is not fresh on p and fresh on q, f remains fresh on q. In imported object structures, non-separate once functions must compute results that satisfy the object structure's invariants when recomputed on any processor. □

The command *import* in Figure 3.25 implements the import operation. It takes an importing processor p and a reference r to the root of an object structure to be imported. The query *last_imported_obj* returns the imported object structure. The *import* command uses the helper function *import_rec_with_map* from Figure 3.26. The function *import_rec_with_map* takes an importing processor p, the handler q of the object structure's root, a reference r of an object to be imported, and a state σ. It returns a reference r'' to the imported object and an updated state σ''. The function *import_rec_with_map* works hand in hand with the function *import_rec_without_map*; they recursively traverse objects. To ensure that no object gets imported twice, the functions use a map w to keep track of the already traversed objects. After traversing an object, w maps that object's reference to the reference of the resulting object in the imported object structure. The command *import* starts the traversal with an empty map. The function *import_rec_with_map* determines whether the object referenced by r has already been traversed, in which case it returns the map value; otherwise, it returns the result of *import_rec_without_map*.

The function *import_rec_without_map* has several steps: a copy step (see o'_0, σ'_0, and w'_0), an attribute values update step (see o'_i), and a result generation step (see r', w', σ'). The attribute values update step uses *import_rec_with_map* recursively to traverse all non-void attribute values of reference type. It updates the copied object accordingly. It sets the attributes $\{a_1, \ldots, a_n\}$ to the references $r_1, \ldots, r_n$ of the recursively imported objects, resulting in o'_n.

```
$last\_imported\_obj\colon STATE \rightarrow REF$
$import\colon STATE \rightarrow PROC \nrightarrow REF \nrightarrow STATE$
    $\sigma.import(p, r)$ **require**
        $\sigma.regions.procs.has(p)$
        $\sigma.heap.refs.has(r)$
        $\sigma.heap.ref\_obj(r).class\_type.is\_exp$
    **axioms**
        $\sigma.import(p, r) = \sigma'$
        $\sigma.import(p, r).last\_imported\_obj = r'$
          **where**
          $w \overset{def}{=}$ **new** $MAP\langle REF, REF\rangle.make$
          $(r', w', \sigma') \overset{def}{=} import\_rec\_with\_map(p, \sigma.handler(r), r, w, \sigma)$
```

Figure 3.25: *STATE*: facade features for importing

$$import\_rec\_with\_map(p, q, r, w, \sigma) \overset{def}{=} (r'', w'', \sigma'')$$

where

$$(r', w', \sigma') \overset{def}{=} import\_rec\_without\_map(p, q, r, w, \sigma)$$

$$r'' \overset{def}{=} \begin{cases} w.val(r) & \text{if } w.keys.has(r) \\ r' & \text{if } \neg w.keys.has(r) \end{cases}$$

$$w'' \overset{def}{=} \begin{cases} w & \text{if } w.keys.has(r) \\ w' & \text{if } \neg w.keys.has(r) \end{cases}$$

$$\sigma'' \overset{def}{=} \begin{cases} \sigma & \text{if } w.keys.has(r) \\ \sigma' & \text{if } \neg w.keys.has(r) \end{cases}$$

$$import\_rec\_without\_map(p, q, r, w, \sigma) \overset{def}{=} (r', w', \sigma')$$

where

$$o'_0 \overset{def}{=} \sigma.ref\_obj(r).copy$$

$$\sigma'_0 \overset{def}{=} \sigma.add\_obj(p, o'_0)$$

$$w'_0 \overset{def}{=} w.add(r, \sigma'_0.ref(o'_0))$$

$$\{a_1, \ldots, a_n\} \overset{def}{=} \{a \mid o'_0.att\_val(a) \in REF \land o'_0.att\_val(a) \neq void\}$$

$$\forall i \in \{1, \ldots, n\}\colon (r'_i, w'_i, \sigma'_i) \overset{def}{=} import\_rec\_with\_map(p, q, o'_0.att\_val(a_i), w'_{i-1}, \sigma'_{i-1})$$

$$\forall i \in \{1, \ldots, n\}\colon o'_i \overset{def}{=} o'_{i-1}.set\_att\_val(a_i, r'_i)$$

$$r' \overset{def}{=} \begin{cases} \sigma'_n.ref(o'_n) & \text{if } \sigma.handler(r) = q \land \{a_1, \ldots, a_n\} \neq \{\} \\ \sigma'_0.ref(o'_0) & \text{if } \sigma.handler(r) = q \land \{a_1, \ldots, a_n\} = \{\} \\ r & \text{otherwise} \end{cases}$$

$$w' \overset{def}{=} \begin{cases} w'_n & \text{if } \sigma.handler(r) = q \land \{a_1, \ldots, a_n\} \neq \{\} \\ w'_0 & \text{if } \sigma.handler(r) = q \land \{a_1, \ldots, a_n\} = \{\} \\ w.add(r, r) & \text{otherwise} \end{cases}$$

$$\sigma' \overset{def}{=} \begin{cases} \sigma'_n.update\_obj(\sigma'_n.ref(o'_0), o'_n) & \text{if } \sigma.handler(r) = q \land \{a_1, \ldots, a_n\} \neq \{\} \\ \sigma'_0 & \text{if } \sigma.handler(r) = q \land \{a_1, \ldots, a_n\} = \{\} \\ \sigma & \text{otherwise} \end{cases}$$

Figure 3.26: *STATE*: facade features for importing

Environments

When a processor p starts executing a feature f, it creates a new variable environment. The processor initializes the formal arguments with the actual arguments $r_1, \ldots, r_n$, the local variables as void, the current entity with the target r_0, and the result entity as void. The command *push_env_with_feature* from Figure 3.27 updates a processor's environment stack accordingly. The query *envs* returns the updated stack, and the command *pop_env* removes the added environment.

```
$envs\colon STATE \to PROC \nrightarrow STACK\langle ENV\rangle$
    $\sigma.envs(p)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.envs(p) = \sigma.store.envs(p)$
$push\_env\_with\_feature\colon STATE \to PROC \nrightarrow FEATURE \to REF$
                                $\to TUPLE\langle REF, \ldots, REF\rangle \nrightarrow STATE$
    $\sigma.push\_env\_with\_feature(p, f, r_0, (r_1, \ldots, r_n))$ **require**
        $\sigma.regions.procs.has(p)$
        $f.formals.count = n$
        $\forall i \in \{0, \ldots, n\}\colon r_i \neq void \Rightarrow \sigma.heap.refs.has(r_i)$
    **axioms**
        $\sigma.push\_env\_with\_feature(p, f, r_0, (r_1, \ldots, r_n)) =$
            $\sigma.set(\sigma.regions, \sigma.heap, \sigma.store.push\_env(p, e))$
          **where**
            $w \overset{def}{=}$ **new** $ENV.make$
                $.update(f.formals(1).name, r_1)\ \ldots$
                $.update(f.formals(n).name, r_n)$
                $.update(f.locals(1).name, void)\ \ldots$
                $.update(f.locals(f.locals.count).name, void)$
                $.update(current.name, r_0)$
            $e \overset{def}{=} \begin{cases} w & \text{if } f \in PROCEDURE \\ w.update(result.name, void) & \text{if } f \in FUNCTION \end{cases}$
$pop\_env\colon STATE \to PROC \nrightarrow STATE$
    $\sigma.pop\_env(p)$ **require**
        $\sigma.regions.procs.has(p)$
        $\neg\sigma.store.envs(p).empty$
    **axioms**
        $\sigma.pop\_env(p) = \sigma.set(\sigma.regions, \sigma.heap, \sigma.store.pop\_env(p))$
```

Figure 3.27: *STATE*: facade features for environments

Writing and reading values

Values can be stored in object attributes, in formal arguments, or in local variables (the result entity is an implicit local variable). The features in Figure 3.28 and Figure 3.29 allow a processor to write and read values in any of these entities. The query *val* takes a processor *p* and a name *n* and it returns the value of *n* in *p*'s execution context, i.e., its top environment and its current object. The command *set_val* writes a value *v* into an entity with name *n*. For both features, it is safe to first check whether the current object has an attribute with name *n* since attribute names and local variable names must be distinct.

$val\colon STATE \rightarrow PROC \nrightarrow NAME \nrightarrow REF \cup PROC$

  $\sigma.val(p, n)$ **require**

    $\sigma.regions.procs.has(p)$

    $\neg\sigma.store.envs(p).empty$

    $e.names.has(current.name)$

    $e.names.has(n) \vee \exists a \in o.class\_type.attributes\colon a.name = n$

   where

    $e \overset{def}{=} \sigma.store.envs(p).top$

    $o \overset{def}{=} \sigma.heap.ref\_obj(e.val(current.name))$

  axioms

$$\sigma.val(p, n) = \begin{cases} \text{if } \exists a \in o.class\_type.attributes\colon a.name = n \\ \quad o.att\_val(a) \\ \text{otherwise} \\ \quad \sigma.store.envs(p).top.val(n) \end{cases}$$

   where

    $o \overset{def}{=} \sigma.heap.ref\_obj(\sigma.store.envs(p).top.val(current.name))$

    $a \overset{def}{=} o.class\_type.feature\_by\_name(n)$

Figure 3.28: *STATE*: facade features for reading values

$set\_val\colon STATE \rightarrow PROC \nrightarrow NAME \nrightarrow REF \cup PROC \nrightarrow STATE$

$\quad \sigma.set\_val(p, n, v)$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\qquad \neg\sigma.store.envs(p).empty \wedge \sigma.store.envs(p).top.names.has(current.name)$

$\qquad v \in REF \wedge v \neq void \Rightarrow \sigma.heap.refs.has(v)$

$\qquad v \in PROC \Rightarrow \sigma.regions.procs.has(v)$

$\quad$ **axioms**

$$\sigma.set\_val(p, n, v) = \begin{cases} \text{if } \exists a \in o.class\_type.attributes\colon a.name = n \\ \quad \sigma.update\_obj(\sigma.heap.ref(o), o.set\_att\_val(a, v)) \\ \text{otherwise} \\ \quad \sigma.set(\sigma.regions, \sigma.heap, s) \end{cases}$$

$\qquad$ **where**

$\qquad o \overset{def}{=} \sigma.heap.ref\_obj(\sigma.store.envs(p).top.val(current.name))$

$\qquad a \overset{def}{=} o.class\_type.feature\_by\_name(n)$

$\qquad s \overset{def}{=} \sigma.store.pop\_env(p).push\_env(p, \sigma.store.envs(p).top.update(n, v))$

Figure 3.29: *STATE*: facade features for writing values

Once routines

Figure 3.30 shows the facade features to query and set the status of once routines. The commands *set_once_func_not_fresh* and *set_once_proc_not_fresh* set a once procedure not fresh; *set_once_rout_fresh* does the reverse. The queries *fresh* and *once_result* return the status of a once routine.

Locks

The facade merges a processor's request queue locks and its call stack locks into two sets. In Figure 3.31, the query *rq_locks* returns the set of all obtained and retrieved request queue locks, and the query *cs_locks* returns the set of all obtained and retrieved call stack locks. The query *rq_locked* returns whether a processor request queue is locked, and the query *passed* returns whether it has currently passed its locks. Figure 3.32 and Figure 3.33 show the facade features to lock request queues, remove obtained request queue locks, unlock request queues, pass locks, and revoke locks.

```
$fresh\colon STATE \rightarrow PROC \nrightarrow ID \nrightarrow BOOLEAN$
    $\sigma.fresh(p, i)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.fresh(p, i) = \sigma.heap.fresh(p, i)$
$once\_result\colon STATE \rightarrow PROC \nrightarrow ID \nrightarrow REF$
    $\sigma.once\_result(p, i)$ **require**
        $\sigma.regions.procs.has(p)$
        $\neg\sigma.heap.fresh(p, i)$
    **axioms**
        $\sigma.once\_result(p, i) = \sigma.heap.once\_result(p, i)$
$set\_once\_func\_not\_fresh\colon STATE \rightarrow PROC \nrightarrow FEATURE \nrightarrow REF \nrightarrow STATE$
    $\sigma.set\_once\_func\_not\_fresh(p, f, r)$ **require**
        $\sigma.regions.procs.has(p)$
        $f \in FUNCTION \wedge f.is\_once$
        $r \neq void \Rightarrow \sigma.heap.refs.has(r)$
    **axioms**
        $\sigma.set\_once\_func\_not\_fresh(p, f, r) =$
            $\sigma.set(\sigma.regions, \sigma.heap.set\_once\_func\_not\_fresh(p, f, r), \sigma.store)$
$set\_once\_proc\_not\_fresh\colon STATE \rightarrow PROC \rightarrow FEATURE \rightarrow STATE$
    $\sigma.set\_once\_proc\_not\_fresh(p, f)$ **require**
        $\sigma.regions.procs.has(p)$
        $f \in PROCEDURE \wedge f.is\_once$
    **axioms**
        $\sigma.set\_once\_proc\_not\_fresh(p, f) =$
            $\sigma.set(\sigma.regions, \sigma.heap.set\_once\_proc\_not\_fresh(p, f), \sigma.store)$
$set\_once\_rout\_fresh\colon STATE \rightarrow PROC \rightarrow FEATURE \nrightarrow STATE$
    $\sigma.set\_once\_rout\_fresh(p, f)$ **require**
        $\sigma.regions.procs.has(p)$
        $f.is\_once$
    **axioms**
        $\sigma.set\_once\_rout\_fresh(p, f) =$
            $\sigma.set(\sigma.regions, \sigma.heap.set\_once\_rout\_fresh(p, f), \sigma.store)$
```

Figure 3.30: *STATE*: facade features for once routines

```
$rq\_locked\colon STATE \to PROC \nrightarrow BOOLEAN$
    $\sigma.rq\_locked(p)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.rq\_locked(p) = \sigma.regions.rq\_locked(p)$
$rq\_locks\colon STATE \to PROC \nrightarrow SET\langle PROC\rangle$
    $\sigma.rq\_locks(p)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.rq\_locks(p) =$
            $\sigma.regions.obtained\_rq\_locks(p).flat \cup \sigma.regions.retrieved\_rq\_locks(p).flat$
$cs\_locks\colon STATE \to PROC \nrightarrow SET\langle PROC\rangle$
    $\sigma.cs\_locks(p)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.cs\_locks(p) =$
            $\{\sigma.regions.obtained\_cs\_lock(p)\} \cup \sigma.regions.retrieved\_cs\_locks(p).flat$
$passed\colon STATE \to PROC \nrightarrow BOOLEAN$
    $\sigma.passed(p)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.passed(p) = \sigma.regions.passed(p)$
```

Figure 3.31: *STATE*: facade queries for locks

$lock\_rqs \colon STATE \to PROC \nrightarrow SET\langle PROC\rangle \nrightarrow STATE$

$\quad \sigma.lock\_rqs(p, \bar{l})$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad \forall x \in \bar{l} \colon \sigma.regions.procs.has(x)$

$\quad\quad \forall x \in \bar{l} \colon \sigma.regions.rq\_locked(x) = false$

$\quad$ **axioms**

$\quad\quad \sigma.lock\_rqs(p, \bar{l}) = \sigma.set(\sigma.regions.lock\_rqs(p, \bar{l}), \sigma.heap, \sigma.store)$

$pop\_obtained\_rq\_locks \colon STATE \to PROC \nrightarrow STATE$

$\quad \sigma.pop\_obtained\_rq\_locks(p)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad \neg\sigma.regions.obtained\_rq\_locks(p).empty$

$\quad\quad \sigma.regions.passed(p) = false$

$\quad$ **axioms**

$\quad\quad \sigma.pop\_obtained\_rq\_locks(p) =$

$\quad\quad\quad \sigma.set(\sigma.regions.pop\_obtained\_rq\_locks(p), \sigma.heap, \sigma.store)$

$unlock\_rq \colon STATE \to PROC \nrightarrow STATE$

$\quad \sigma.unlock\_rq(p)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad \sigma.regions.rq\_locked(p) = true$

$\quad\quad \forall q \in \sigma.regions.procs \colon \neg\sigma.regions.obtained\_rq\_locks(q).flat.has(p)$

$\quad$ **axioms**

$\quad\quad \sigma.unlock\_rq(p) = \sigma.set(\sigma.regions.unlock\_rq(p), \sigma.heap, \sigma.store)$

Figure 3.32: *STATE*: facade commands for locking and unlocking

$\mathit{pass\_locks}\colon \mathit{STATE} \to \mathit{PROC} \nrightarrow \mathit{PROC} \nrightarrow \mathit{TUPLE}\langle \mathit{SET}\langle \mathit{PROC}\rangle, \mathit{SET}\langle \mathit{PROC}\rangle\rangle$
$\quad\quad \nrightarrow \mathit{STATE}$

$\quad \sigma.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c}))$ **require**

$\quad\quad \sigma.\mathit{regions.procs.has}(p) \wedge \sigma.\mathit{regions.procs.has}(q)$

$\quad\quad \forall x \in \overline{l_r}\colon\ \sigma.\mathit{regions.obtained\_rq\_locks}(p).\mathit{flat.has}(x) \vee$
$\quad\quad\quad \sigma.\mathit{regions.retrieved\_rq\_locks}(p).\mathit{flat.has}(x)$

$\quad\quad \forall x \in \overline{l_c}\colon\ x = \sigma.\mathit{regions.obtained\_cs\_lock}(p) \vee$
$\quad\quad\quad \sigma.\mathit{regions.retrieved\_cs\_locks}(p).\mathit{flat.has}(x)$

$\quad\quad \neg\overline{l_r}.\mathit{empty} \vee \neg\overline{l_c}.\mathit{empty} \Rightarrow \neg\sigma.\mathit{regions.passed}(p)$

$\quad$ **axioms**

$\quad\quad \sigma.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})) =$
$\quad\quad\quad \sigma.\mathit{set}(\sigma.\mathit{regions.pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})), \sigma.\mathit{heap}, \sigma.\mathit{store})$

$\mathit{revoke\_locks}\colon \mathit{STATE} \to \mathit{PROC} \nrightarrow \mathit{PROC} \nrightarrow \mathit{STATE}$

$\quad \sigma.\mathit{revoke\_locks}(p, q)$ **require**

$\quad\quad \sigma.\mathit{regions.procs.has}(p) \wedge \sigma.\mathit{regions.procs.has}(q)$

$\quad\quad \neg\sigma.\mathit{regions.retrieved\_rq\_locks}(q).\mathit{empty} \wedge$
$\quad\quad\quad \neg\sigma.\mathit{regions.retrieved\_cs\_locks}(q).\mathit{empty}$

$\quad\quad \sigma.\mathit{regions.retrieved\_rq\_locks}(q).\mathit{top} \subseteq$
$\quad\quad\quad \sigma.\mathit{regions.obtained\_rq\_locks}(p).\mathit{flat} \cup \sigma.\mathit{regions.retrieved\_rq\_locks}(p).\mathit{flat}$

$\quad\quad \sigma.\mathit{regions.retrieved\_cs\_locks}(q).\mathit{top} \subseteq$
$\quad\quad\quad \{\sigma.\mathit{regions.obtained\_cs\_lock}(p)\} \cup \sigma.\mathit{regions.retrieved\_cs\_locks}(p).\mathit{flat}$

$\quad\quad \sigma.\mathit{regions.retrieved\_rq\_locks}(q).\mathit{top} \cup \sigma.\mathit{regions.retrieved\_cs\_locks}(q).\mathit{top} \neq \{\} \Rightarrow$
$\quad\quad\quad \sigma.\mathit{regions.passed}(p) = \mathit{true}$

$\quad\quad \sigma.\mathit{regions.passed}(q) = \mathit{false}$

$\quad$ **axioms**

$\quad\quad \sigma.\mathit{revoke\_locks}(p, q) = \sigma.\mathit{set}(\sigma.\mathit{regions.revoke\_locks}(p, q), \sigma.\mathit{heap}, \sigma.\mathit{store})$

Figure 3.33: *STATE*: facade commands for passing and revoking locks

Notation

This section presents a notation for states. A state has four parts: the locks, the objects, the once status, and the environments.

- The locks part shows for each processor the stack of obtained request queue locks (orq), the stack of retrieved request queue locks (rrq), and the stack of retrieved call stack locks (rcs). The notation has two flags to indicate when a processor's request queue is locked or unlocked. It also has a flag to indicate when a processor passed its locks; in absence of this flag, the locks are not passed.
- The objects part shows for each processor the handled objects with their references. It also shows the content of the objects. For objects other than arrays or objects of basic class type, it shows a list of attribute values, omitting the ones that are void. For objects of basic type, it shows the basic value; for arrays, it shows the cells of the array.
- The once status part shows the status of each once routine that is not fresh. A once routine can be not fresh either with respect to a subset of processors or with respect to all processors. In the first case, the once status part shows the once routine in connection with each processor in the subset. In the second case, it uses an indicator to denote all processors.
- The environments part shows the environments for each processor.

For example, consider the state of a system with three processors:

locks:
p_1 :: orq: $(\{p_2\},\{p_3\})$ rrq: $(\{\},\{\})$ rcs: $(\{\},\{\})$ locked passed
p_2 :: orq: $()$ rrq: $()$ rcs: $()$ locked
p_3 :: orq: $()$ rrq: $(\{p_2,p_3\})$ rcs: $(\{p_1\})$ locked
objects:
p_1 :: $r_1 \rightarrow o_1$
p_2 :: $r_2 \rightarrow o_2[[r_3,r_3],[r_3,r_3]], r_3 \rightarrow o_3, r_4 \rightarrow o_4(1)$
p_3 :: $r_5 \rightarrow o_5(id \rightarrow r_6), r_6 \rightarrow o_6(2)$
once status:
p_2 :: $\{APPLICATION\}.id \rightarrow r_4$
all :: $\{APPLICATION\}.initialize$
environments:
p_1 :: $node \rightarrow r_2$, **Current** $\rightarrow r_1$ / $node \rightarrow r_5$, **Current** $\rightarrow r_1$
p_2 ::
p_3 :: $root \rightarrow r_1$, **Current** $\rightarrow r_5$, **Result** $\rightarrow void$

The request queues of all processors are locked. Processor p_1 has a stack of obtained request queue locks with two items. The set $\{p_2\}$ is on the bottom of the stack, and the set $\{p_3\}$ is on the top. Processor p_1's stack of retrieved request queue locks and retrieved call stack locks each consist of two empty sets. Processor p_1 passed its locks, and p_3's entry shows that these locks have been passed from p_1 to p_3. Processor p_2 does not hold any locks, except its own call stack lock. Processor p_1 handles an object o_1 referenced by r_1 while processors p_2 and p_3 each handle multiple objects. Object o_5 has an attribute *id* referencing object o_6. Objects o_6 and o_4 are integers with values 2 and 1. Object o_2 is a two dimensional array with 2×2 cells, where each of the cells references object o_3. The once function *id* of class *APPLICATION* is not fresh with the value r_4 on processor p_2. The once procedure *initialize* is not fresh on all processors in the system. Processor p_1 has a stack with two environments, the environment on the left at the bottom of the stack, and the environment on the right at the top. Processor p_2 has no environments. Processor p_3 has one with three mappings: the entity *root* has the value r_1, the current entity has the value r_5, and the result entity has the void value.

3.3 Execution specification

This section uses a structural operational semantics [179] to describe a concurrent program's interleaved steps in terms of the program elements, e.g., command instructions or query expressions. For this purpose, it introduces a number of runtime mechanisms, used by the processors to execute program elements.

3.3.1 Executions

An *execution* is a sequence of *configurations*. Each configuration of the form $\langle p_1 :: s_{p1} \mid \ldots \mid p_n :: s_{pn}, \sigma \rangle$ is an execution snapshot, consisting of the *schedule* and the *state* σ. The schedule consists of the call stacks and the request queues of $p_1, \ldots, p_n$, and the state σ is of type *STATE*. The call stack and the request queue of a processor are also known as the *action queue* [173] of the processor. The commutative and associative parallel operator $\mid$ keeps the processors' action queues apart. Each action queue contains a statement sequence of type *STATEMENT_SEQUENCE*, executed in FIFO order. The beginning of the action queue contains the statements for the features that are being executed at the moment. The tail of the action queue is the request queue of the processor. A call stack lock is the right to add statements to the beginning of the action queue. Similarly, a request queue lock is the right to add statements to the end of the action queue. Statements are either instructions, i.e., program elements, or operations, i.e., runtime elements. As for instructions (see Figure 3.2), Figure 3.35 structurally defines the types of the operations.

Transition rules describe the transitions that take the system from a *start configuration* into a *result configuration*. The following well-known transition rule for parallelism allows a processor to perform one step in parallel with other processors:

PARALLELISM

$$\frac{\Gamma \vdash \langle P, \sigma\rangle \rightarrow \langle P', \sigma'\rangle}{\Gamma \vdash \langle P \mid Q, \sigma\rangle \rightarrow \langle P' \mid Q, \sigma'\rangle}$$

In the rule, the typing environment Γ (see Section 3.1.8) provides static information about the program under execution.

3.3.2 Initial configuration

A SCOOP program *prg* of type *PROGRAM* defines a root class and a root procedure, which has no formal arguments and no precondition. In the beginning, the runtime generates a bootstrap processor p and a root processor q with a root object. The initial state σ''' in Figure 3.34 reflects this.

$$\sigma \overset{def}{=} \textbf{new}\ \mathit{STATE.make}$$

$$\sigma' \overset{def}{=} \sigma.\mathit{add\_proc}(\sigma.\mathit{new\_proc})\quad p \overset{def}{=} \sigma'.\mathit{last\_added\_proc}$$

$$\sigma'' \overset{def}{=} \sigma'.\mathit{add\_proc}(\sigma'.\mathit{new\_proc})\quad q \overset{def}{=} \sigma''.\mathit{last\_added\_proc}$$

$$\sigma''' \overset{def}{=} \sigma''.\mathit{add\_obj}(q, \sigma''.\mathit{new\_obj}(\mathit{prg.settings.root\_class}))\quad r \overset{def}{=} \sigma'''.\mathit{last\_added\_obj}$$

$$\begin{aligned}\langle p :: {} & \texttt{lock}(\{q\});\\ & \texttt{call}(r, \mathit{prg.settings.root\_procedure}, ());\\ & \texttt{issue}(q, \texttt{unlock\_rq});\\ & \texttt{pop\_obtained\_locks} \mid q ::, \sigma'''\rangle\end{aligned}$$

Figure 3.34: Initial configuration of a program *prg*

The bootstrap processor p first locks the request queue of the root processor q. It then asks the root processor to execute the root procedure on the root object and unlock thereafter. Finally, it removes the request queue lock from its stack of obtained request queue locks, as shown in the initial configuration from Figure 3.34. In a nutshell, the $\texttt{lock}(\{q\})$ operation locks the request queue of q. The $\texttt{call}(r, \mathit{prg.settings.root\_procedure}, ())$ operation asks r's handler to execute the

```
OPERATION ≜ ISSUE | RESULT | NOTIFY | WAIT | EVAL | LOCK |
   UNLOCK_RQ | POP_OBTAINED_LOCKS | WRITE | READ | PROVIDED |
   NOP | CALL | APPLY | CHECK_PRE_AND_LOCK | EXECUTE_BODY |
   SET_NOT_FRESH | CHECK_POST_AND_INV | RETURN ;
ISSUE ≜ issue TUPLE⟨PROC, STATEMENT_SEQUENCE⟩ ;
RESULT ≜ result TUPLE⟨CHANNEL, REF⟩ ;
NOTIFY ≜ notify TUPLE⟨CHANNEL⟩ ;
WAIT ≜ wait TUPLE⟨CHANNEL⟩ ;
EVAL ≜ eval TUPLE⟨CHANNEL, EXPRESSION⟩ ;
LOCK ≜ lock TUPLE⟨SET⟨PROC⟩⟩ ;
UNLOCK_RQ ≜ unlock_rq ;
POP_OBTAINED_LOCKS ≜ pop_obtained_locks ;
WRITE ≜ write TUPLE⟨ENTITY, (REF | PROC), BOOLEAN⟩ ;
READ ≜ read TUPLE⟨ENTITY, CHANNEL⟩ ;
PROVIDED ≜
   provided (REF | BOOLEAN) then
      STATEMENT_SEQUENCE
   else
      STATEMENT_SEQUENCE
   end ;
NOP ≜ nop ;
CALL ≜ call TUPLE⟨REF, FEATURE, TUPLE⟨REF, ..., REF⟩⟩ |
   call TUPLE⟨CHANNEL, REF, FEATURE, TUPLE⟨REF, ..., REF⟩⟩ ;
APPLY ≜ apply TUPLE⟨CHANNEL, REF, FEATURE, TUPLE⟨REF, ..., REF⟩,
    PROC, TUPLE⟨SET⟨PROC⟩, SET⟨PROC⟩⟩⟩ ;
CHECK_PRE_AND_LOCK ≜
   check_pre_and_lock TUPLE⟨FEATURE, SET⟨PROC⟩⟩ ;
EXECUTE_BODY ≜ execute_body TUPLE⟨FEATURE⟩ ;
SET_NOT_FRESH ≜ set_not_fresh TUPLE⟨FEATURE⟩ ;
CHECK_POST_AND_INV ≜ check_post_and_inv TUPLE⟨FEATURE⟩ ;
RETURN ≜ return TUPLE⟨CHANNEL, PROC⟩ |
   return TUPLE⟨CHANNEL, REF, PROC⟩ ;
```

Figure 3.35: *OPERATION*

root procedure on r. The `unlock_rq` operation unlocks the request queue of the processor that executes the operation. The `issue`$(q,$ `unlock_rq`$)$ operation adds the `unlock_rq` operation to q's action queue. The `pop_obtained_locks` operation removes the top element from the stack of obtained request queue locks.

3.3.3 Issuing mechanism

The issuing mechanism allows a processor to add statements to an action queue. To issue statements s_w to a processor q, processor p executes an `issue`(q, s_w) operation. Figure 3.36 shows the transition rules.

Issue (non-separate)

$$\frac{\begin{array}{l} q = p \\ \neg\sigma.passed(p) \wedge \sigma.cs\_locks(p).has(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p, \sigma\rangle \to \langle p :: s_w; s_p, \sigma\rangle}$$

Issue (separate)

$$\frac{\begin{array}{l} q \neq p \wedge \neg(\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\ \neg\sigma.passed(p) \wedge \sigma.rq\_locks(p).has(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \to \langle p :: s_p \mid q :: s_q; s_w, \sigma\rangle}$$

Issue (separate callback)

$$\frac{\begin{array}{l} q \neq p \wedge (\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\ \neg\sigma.passed(p) \wedge \sigma.cs\_locks(p).has(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \to \langle p :: s_p \mid q :: s_w; s_q, \sigma\rangle}$$

Figure 3.36: Issuing mechanism transition rules

Processor p can issue statements to itself or to another processor. In the non-separate case, i.e., $p = q$, p adds the statements to the beginning of its action queue, i.e., its call stack, requiring that p holds its call stack lock. In the separate case, i.e., $p \neq q$, p adds the statements to q's action queue. If q claims a lock on p, indicating a separate callback, p uses q's call stack lock to add s_w to the beginning of q's action queue for immediate execution. On the other hand, if q does not claim a lock on p, p adds the statements to the end of q's action queue, requiring that p holds q's request queue lock.

3.3.4 Notification mechanism

Processors communicate over channels of type *CHANNEL*. A processor p creates a new channel a when using a transition rule with "a is fresh" in the premise. A processor q can then use a to communicate with p using the transition rules from Figure 3.37. The send a notification with value r over a, q adds the $\texttt{result}(a, r)$ operation into its action queue. To send a notification without a value over a, q adds $\texttt{notify}(a)$. Processor p executes $\texttt{wait}(a)$ to wait for a notification on a. It continues once the notification on a is ready; in case a carries a value, it substitutes all references to the channel (see $a.data$) with the notification value.

Wait for result (non-separate)

$$\frac{}{\Gamma \vdash \langle p :: \texttt{result}(a, r); s_w; \texttt{wait}(a); s_p, \sigma\rangle \to \langle p :: s_w; s_p[r/a.data], \sigma\rangle}$$

Wait for notify (non-separate)

$$\frac{}{\Gamma \vdash \langle p :: \texttt{notify}(a); s_w; \texttt{wait}(a); s_p, \sigma\rangle \to \langle p :: s_w; s_p, \sigma\rangle}$$

Wait for result (separate)

$$\frac{}{\Gamma \vdash \langle p :: s_w; \texttt{wait}(a); s_p \mid q :: \texttt{result}(a, r); s_q, \sigma\rangle \to \langle p :: s_w; s_p[r/a.data] \mid q :: s_q, \sigma\rangle}$$

Wait for notify (separate)

$$\frac{}{\Gamma \vdash \langle p :: s_w; \texttt{wait}(a); s_p \mid q :: \texttt{notify}(a); s_q, \sigma\rangle \to \langle p :: s_w; s_p \mid q :: s_q, \sigma\rangle}$$

Figure 3.37: Notification mechanism transition rules

To ensure that each `wait` operation can be resolved with exactly one `result` or `notify` operation, the following must hold:

- For each $\texttt{wait}(a)$ operation there must be either exactly one $\texttt{result}(a, r)$ or exactly one $\texttt{notify}(a)$ operation.
- For each $\texttt{result}(a, r)$ or $\texttt{notify}(a)$ operation there must be exactly one $\texttt{wait}(a)$ operation.
- Each $\texttt{result}(a, r)$ or $\texttt{notify}(a)$ operation precedes the $\texttt{wait}(a)$ operation. A statement s_1 precedes a statement s_2 if and only if s_1 appears earlier than s_2 in the same action queue or s_1 and s_2 appear in different action queues.

The notification mechanism is inspired by the π-calculus [156], where the expression $c(x).P$ denotes a process that is waiting for a notification sent on a channel c. Once the process receives the notification, it binds the value of the notification to the variable x and continues with the expression P. The notification comes from a process that executes $\bar{c}y.Q$ to emit the value y on the channel c before executing Q.

3.3.5 Expression evaluation mechanism

An expression is a literal, an entity, or a query call. A query call can contain actual arguments that are expressions themselves. The expression evaluation mechanism allows a processor to evaluate any expression e and post the result on a channel a by executing `eval`(a, e). The `eval`(a, e) operation results in a `result`(a, r) operation, and executing `wait`(a) enables a processor to use the evaluation result. The following sections overload the `eval` operation for the different expression types.

3.3.6 Locking and unlocking mechanism

A processor p about to execute a feature must first obtain the request queue locks of the suppliers $\{q_1, \ldots, q_m\}$ by executing the `lock`$(\{q_1, \ldots, q_m\})$ operation from Figure 3.38. The `lock` operation requires that none of the request queues is already locked. Once p is done with the execution of the feature, it asks $\{q_1, \ldots, q_m\}$ to unlock their request queues once they are done with the issued statements. For this purpose, p issues the `unlock_rq` operation to processors $\{q_1, \ldots, q_m\}$. The `unlock_rq` operation requires that the request queue is indeed locked and that no processor claims the request queue lock. Once p issued the `unlock_rq` operations, it removes $\{q_1, \ldots, q_m\}$ from its stack of obtained request queue locks using the `pop_obtained_locks` operation; this ensures that the `unlock_rq` operations can proceed.

3.3.7 Writing and reading mechanism

Figure 3.39 describes operations to write and read values in entities of a processor's execution context, i.e., its top environment and its current object. A processor p executes the `write`(b, v, ce) operation to set the value of an entity b to v. If ce is true and v references an expanded object on p, the operation copies the expanded object first. Similarly, processor p execute the `read`(b, a) operation to read a value of an entity b and post it on a.

Lock

$$\frac{\begin{array}{l}\neg\exists q_i \in \{q_1,\ldots,q_m\}\colon \sigma.rq\_locked(q_i)\\ \sigma' \overset{def}{=} \sigma.lock\_rqs(p,\{q_1,\ldots,q_m\})\end{array}}{\Gamma \vdash \langle p :: \texttt{lock}(\{q_1,\ldots,q_m\}); s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Unlock request queue

$$\frac{\begin{array}{l}\sigma.rq\_locked(p)\\ \forall q \in \sigma.procs\colon \neg\sigma.rq\_locks(q).has(p)\\ \sigma' \overset{def}{=} \sigma.unlock\_rq(p)\end{array}}{\Gamma \vdash \langle p :: \texttt{unlock\_rq}; s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Pop obtained locks

$$\frac{\sigma' \overset{def}{=} \sigma.pop\_obtained\_rq\_locks(p)}{\Gamma \vdash \langle p :: \texttt{pop\_obtained\_locks}; s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Figure 3.38: Locking and unlocking mechanism transition rules

Write

$$\frac{\begin{array}{l}(\sigma', v') \overset{def}{=} \begin{cases}\text{if}\\ \quad ce\wedge\\ \quad v \in REF \wedge v \neq void \wedge \sigma.ref\_obj(v).class\_type.is\_exp\wedge\\ \quad \sigma.handler(v) = p\\ \text{then}\\ \quad (\sigma^*, \sigma^*.last\_added\_obj)\\ \quad\quad \textbf{where}\\ \quad\quad\quad \sigma^* \overset{def}{=} \sigma.add\_obj(p, \sigma.ref\_obj(v).copy)\\ \text{otherwise}\\ \quad (\sigma, v)\end{cases}\\ \sigma'' \overset{def}{=} \sigma'.set\_val(p, b.name, v')\end{array}}{\Gamma \vdash \langle p :: \texttt{write}(b, v, ce); s_p, \sigma\rangle \to \langle p :: s_p, \sigma''\rangle}$$

Read

$$\frac{}{\Gamma \vdash \langle p :: \texttt{read}(b, a); s_p, \sigma\rangle \to \langle p :: s_p[\sigma.val(p, b.name)/a.data], \sigma\rangle}$$

Figure 3.39: Writing and reading mechanism transition rules

3.3.8 Flow control mechanism

A processor p executing the `provided` v `then` s_t `else` s_f `end` operation executes s_t if v is true and s_f if v is false. The transition rules in Figure 3.40 cover both cases. The value v can either be an instance of the ADT *BOOLEAN* or a reference pointing to an object with class type *boolean*. If v is an instance of *BOOLEAN*, then p determines which instance v is, i.e., *true* or *false*. If v is a reference, then p gets the referenced object and sees which boolean value it represents by evaluating the attribute *item*. The branches s_t and s_f can be empty; the `nop` operation represents such empty statement sequences.

Conditional (true)

$$\frac{y \overset{def}{=} \begin{cases} v & \text{if } v \in BOOLEAN \\ \sigma.ref\_obj(v).att\_val(item) & \text{if } v \in REF \wedge \sigma.ref\_obj(v).class\_type = boolean \\ false & \text{otherwise} \end{cases} \quad y = true}{\Gamma \vdash \langle p :: \texttt{provided}\ v\ \texttt{then}\ s_t\ \texttt{else}\ s_f\ \texttt{end}; s_p, \sigma\rangle \rightarrow \langle p :: s_t; s_p, \sigma\rangle}$$

Conditional (false)

$$\frac{y \overset{def}{=} \begin{cases} v & \text{if } v \in BOOLEAN \\ \sigma.ref\_obj(v).att\_val(item) & \text{if } v \in REF \wedge \sigma.ref\_obj(v).class\_type = boolean \\ true & \text{otherwise} \end{cases} \quad y = false}{\Gamma \vdash \langle p :: \texttt{provided}\ v\ \texttt{then}\ s_t\ \texttt{else}\ s_f\ \texttt{end}; s_p, \sigma\rangle \rightarrow \langle p :: s_f; s_p, \sigma\rangle}$$

Skip

$$\frac{}{\Gamma \vdash \langle p :: \texttt{nop}; s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma\rangle}$$

Figure 3.40: Flow control mechanism transition rules

3.3.9 Entity and literal expressions

To evaluate an entity expression e, a processor uses the overloaded `eval`(a, e) operation as specified in Figure 3.41: the processor executes the `read` operation and posts a notification with the read value on a.

To evaluate a void literal, the processor takes the void reference, as can be seen in Figure 3.41. To evaluate a non-void literal expression, the processor creates a new object of the literal class type with the corresponding value. For this purpose, it uses the function *obj* (see Section 3.1.7). Since every literal is non-separate, the

Entity expression

$$\frac{\begin{array}{l} e \in \mathit{ENTITY} \\ a' \text{ is fresh} \end{array}}{\Gamma \vdash \langle p :: \texttt{eval}(a, e); s_p, \sigma\rangle \to \langle p :: \texttt{read}(e, a'); \texttt{result}(a, a'.data); s_p, \sigma\rangle}$$

Literal expression

$$\frac{\begin{array}{l} e \in \mathit{LITERAL} \\ \sigma' \overset{def}{=} \begin{cases} \sigma & \text{if } e = \textbf{Void} \\ \sigma.add\_obj(p, obj(e)) & \text{otherwise} \end{cases} \\ r \overset{def}{=} \begin{cases} void & \text{if } e = \textbf{Void} \\ \sigma'.last\_added\_obj & \text{otherwise} \end{cases} \end{array}}{\Gamma \vdash \langle p :: \texttt{eval}(a, e); s_p, \sigma\rangle \to \langle p :: \texttt{result}(a, r); s_p, \sigma'\rangle}$$

Figure 3.41: Entity and literal expressions transition rules

processor creates the new object on itself. The reference r to the new object is the result of the evaluation.

3.3.10 Feature calls

A feature call is either a command call in a command instruction or a query call in a query expression. Figure 3.42 and Figure 3.43 show the transition rules for both cases. In each case, a processor p first evaluates the target expression e_0 and all argument expressions $e_1, \dots, e_n$. It then uses the result of these evaluations to perform the actual feature call by executing the `call` operation. For queries, p creates a new channel a' and uses it in the `call` operation to receive the query result.

Command instruction

$$\frac{\forall i \in \{0, \dots, n\} \colon a_i \text{ is fresh}}{\begin{array}{l} \Gamma \vdash \langle p :: e_0.f(e_1, \dots, e_n); s_p, \sigma\rangle \to \\ \quad \langle p :: \texttt{eval}(a_0, e_0); \texttt{eval}(a_1, e_1); \dots; \texttt{eval}(a_n, e_n); \\ \qquad \texttt{wait}(a_0); \texttt{wait}(a_1); \dots; \texttt{wait}(a_n); \\ \qquad \texttt{call}(a_0.data, f, (a_1.data, \dots, a_n.data)); \\ \qquad s_p, \sigma\rangle \end{array}}$$

Figure 3.42: Command instructions transition rules

QUERY EXPRESSION

$$\frac{\forall i \in \{0, \ldots, n\}\colon a_i \text{ is fresh} \quad a' \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \mathtt{eval}(a, e_0.f(e_1, \ldots, e_n)); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \mathtt{eval}(a_0, e_0); \mathtt{eval}(a_1, e_1); \ldots; \mathtt{eval}(a_n, e_n); \\ \qquad \mathtt{wait}(a_0); \mathtt{wait}(a_1); \ldots; \mathtt{wait}(a_n); \\ \qquad \mathtt{call}(a', a_0.data, f, (a_1.data, \ldots, a_n.data)); \\ \qquad \mathtt{result}(a, a'.data); \\ \qquad s_p, \sigma\rangle\end{array}}$$

Figure 3.43: Query expressions transition rules

According to the informal description, p passes all its locks, including a lock on itself, if it binds a controlled actual argument to an attached formal argument of reference type. This lock passing condition is static, allowing programmers to statically reason about the code and supporting compilers to optimize the call code. However, the condition is not precise enough to capture all situations where the supplier needs the locks of the client. Consider the following classes:

```
class A create make feature
  b: separate B
  c: separate C

  make
    do
      create b.make; create c.make
      f (b)
    end

  f (b: separate B)
    do
      g (c)
    end

  g (c: separate C)
    local
      l: separate B
    do
      l := c.h (b)
    end
end
```

```
class B create make feature
  make
    do end
end

class C create make feature
  make
    do end

  h (b: separate B): separate B
    do
      Result := b
    end
end
```

The program creates three processors, *a*, *b*, and *c*. Processor *a* handles an object of type *A*. Likewise, processor *b* handles an object of type *B*, and processor *c* handles an object of type *C*. In feature *f*, *a* obtains the request queue lock of *b*. It then executes feature *g* and calls a query *c*.*h* (*b*). The supplier requires the request queue lock of *b*, currently held by *a*. However, the condition from the informal description does not trigger *a* to pass its locks because the expression *b* is not controlled in *g*. A dynamic condition solves this flaw: a client *p* passes its locks whenever the supplier *q* needs any of its locks to execute the called feature. Processor *p* passes all its locks, including its call stack lock but not necessarily its request queue lock. The call stack lock permits *q* to perform a separate callback to *p*, during which *q* must pass back *p*'s call stack lock along with the rest of its locks. The following clarification summarizes these changes.

Clarification 3.5 (Lock passing). During a feature call, the client *p* passes its locks to the supplier *q* in two cases: (1) *p* holds a lock on a processor handling an argument of attached reference type; or (2) the feature call is a separate callback. In both cases, *p* passes all its locks, including its call stack lock. □

The supplier only needs locks on processors handling arguments of attached reference type. An argument of detachable type indicates that the handler must not be locked (selective locking mechanism), and an expanded argument has a copy semantics. The supplier receives a copied or imported object and hence must not lock the handler. For the latter case, the informal description does not precisely state when to copy or import arguments of expanded type. Consider the following code:

```
class A create make feature
  make
    local
      b: separate B
      c: C
    do
      create b.make ; create c.make
      f (b, c)
    end

  f (b: separate B; c: C)
    do
      b.g (c)
      c.h
    end
end

class B create make feature
  make
    do end

  g (c: C)
    do end
end

expanded class C create make feature
  i: INTEGER

  make
    do
      i := 0
    end

  h
    do
      i := i + 1
    end
end
```

In feature f, the root processor a initializes an expanded object c and passes it to processor b in $b.g\ (c)$. Then a modifies c. Since the feature call is asynchronous, b can execute g before, while, or after a modifies c. If it is b's responsibility to import c, a race condition occurs because b can import c before or after a modifies c, or worse yet, while a modifies it. To avoid this issue, a client must copy or import expanded objects to a supplier at the moment of the call.

Clarification 3.6 (Copy and import). During a feature call, the client must copy or import expanded objects to a supplier. □

Both clarifications are reflected in Figure 3.44 and Figure 3.45. The `call` operation is overloaded for commands and for queries. Both variants take the reference to the target r_0, the feature f to be called, and the references $(r_1, \ldots, r_n)$ for the evaluated argument expressions. The variant for queries additionally takes a channel a to be used for the query result. Processor p first determines the target handler q, the actual arguments $r'_1, \ldots, r'_n$, and the locks $\bar{l}$ to be passed. The actual arguments $r'_1, \ldots, r'_n$ correspond to $r_1, \ldots, r_n$ where expanded objects are copied or imported to q. The tuple $\bar{l}$ contains p's request queue and call stack locks in case q needs one of p's locks (see first condition) or in case of a separate callback (see second condition); otherwise $\bar{l}$ is empty (see third condition). Processor p then issues a feature request `apply`$(a, r_0, f, (r'_1, \ldots, r'_n), p, l)$ to q. The `issue` operation determines the nature of the call (non-separate, separate, or separate callback) and places the feature request accordingly. The feature request contains a channel a for the communication between p and q. For queries, p uses the channel to retrieve the query result. For commands, p uses the channel to wait for passed locks to return. Processor p does not pass the locks right away because q might not be ready yet. Instead, q retrieves the locks while it applies the feature request and returns them after completing the request.

Call feature (command)

$$q \overset{def}{=} \sigma.handler(r_0)$$

$$\bar{l} \overset{def}{=} \begin{cases} \text{if} \\ \quad q \neq p \wedge \exists i \in \{1, \ldots, n\}, z, c \colon \Gamma \vdash f.formals(i) : (!, z, c) \wedge \\ \qquad \sigma.ref\_obj(r_i).class\_type.is\_ref \wedge \\ \qquad (\neg\sigma.passed(p) \wedge (\sigma.rq\_locks(p) \cup \sigma.cs\_locks(p)).has(\sigma.handler(r_i))) \\ \text{then} \\ \qquad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \text{if} \\ \quad q \neq p \wedge (\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\ \text{then} \\ \qquad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \text{otherwise} \\ \qquad (\{\}, \{\}) \end{cases}$$

$$\sigma'_0 \overset{def}{=} \sigma$$

$$\forall i \in \{1, \ldots, n\} \colon (\sigma'_i, r'_i) \overset{def}{=}$$

$$\begin{cases} \text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) \neq q \\ \quad (\sigma^*, \sigma^*.last\_imported\_obj) \\ \qquad \textbf{where} \\ \qquad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.import(q, r_i) \\ \text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) = q \\ \quad (\sigma^*, \sigma^*.last\_added\_obj) \\ \qquad \textbf{where} \\ \qquad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.add\_obj(q, \sigma'_{i-1}.ref\_obj(r_i).copy) \\ \text{otherwise} \\ \quad (\sigma'_{i-1}, r_i) \end{cases}$$

a is fresh

$$\Gamma \vdash \langle p :: \texttt{call}(r_0, f, (r_1, \ldots, r_n)); s_p, \sigma\rangle \rightarrow$$
$$\langle p :: \texttt{issue}(q, \texttt{apply}(a, r_0, f, (r'_1, \ldots, r'_n), p, \bar{l}));$$
$$\texttt{provided}\ \bar{l} \neq (\{\}, \{\})\ \texttt{then wait}(a)\ \texttt{else nop end};$$
$$s_p, \sigma'_n\rangle$$

Figure 3.44: Command calls transition rules

Call feature (query)

$$q \overset{def}{=} \sigma.handler(r_0)$$

$$\bar{l} \overset{def}{=} \begin{cases} \text{if} \\ \quad q \neq p \wedge \exists i \in \{1, \ldots, n\}, z, c \colon \Gamma \vdash f.formals(i) : (!, z, c) \wedge \\ \qquad \sigma.ref\_obj(r_i).class\_type.is\_ref \wedge \\ \qquad (\neg \sigma.passed(p) \wedge (\sigma.rq\_locks(p) \cup \sigma.cs\_locks(p)).has(\sigma.handler(r_i))) \\ \text{then} \\ \qquad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \text{if} \\ \quad q \neq p \wedge (\sigma.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p)) \\ \text{then} \\ \qquad (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) \\ \text{otherwise} \\ \qquad (\{\}, \{\}) \end{cases}$$

$$\sigma'_0 \overset{def}{=} \sigma$$

$$\forall i \in \{1, \ldots, n\} \colon (\sigma'_i, r'_i) \overset{def}{=}$$

$$\begin{cases} \text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) \neq q \\ \quad (\sigma^*, \sigma^*.last\_imported\_obj) \\ \qquad \textbf{where} \\ \qquad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.import(q, r_i) \\ \text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) = q \\ \quad (\sigma^*, \sigma^*.last\_added\_obj) \\ \qquad \textbf{where} \\ \qquad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.add\_obj(q, \sigma'_{i-1}.ref\_obj(r_i).copy) \\ \text{otherwise} \\ \quad (\sigma'_{i-1}, r_i) \end{cases}$$

$$\frac{}{\begin{array}{l} \Gamma \vdash \langle p :: \mathtt{call}(a, r_0, f, (r_1, \ldots, r_n)); s_p, \sigma \rangle \rightarrow \\ \quad \langle p :: \mathtt{issue}(q, \mathtt{apply}(a, r_0, f, (r'_1, \ldots, r'_n), p, \bar{l})); \mathtt{wait}(a); s_p, \sigma'_n \rangle \end{array}}$$

Figure 3.45: Query calls transition rules

3.3.11 Feature applications

A feature request for a processor p is an `apply`$(a, r_0, f, (r_1, \ldots, r_n), q, \bar{l})$ operation in p's action queue. Processor p uses channel a to communicate with the client q after the execution of f. The reference r_0 points to the target of the call; $r_1, \ldots, r_n$ point to the actual arguments. The tuple $\bar{l}$ contains the locks to be passed from q to p. Three cases exist:

- The feature f is a non-once routine or a fresh once routine.
- The feature f is a not fresh once routine.
- The feature f is an attribute.

Figure 3.46 shows the transition rule for the first case. Processor p can only apply f on one of its own objects. In case f is a once routine, p first sets f's once status to not fresh (see σ'). It then receives the passed locks from q and creates a new environment with the actual arguments $(r_1, \ldots, r_n)$ (see σ''). Next, p locks the request queues of its suppliers and checks the precondition (see `check_pre_and_lock`). For each feature call target in f, p must either hold a call stack lock or a request queue lock on the target's handler. Call stack locks are necessary for non-separate calls and separate callbacks, and request queue locks are necessary for all other (separate) calls. Processor p thus defines two sets of required locks (see $\overline{g}_{required\_cs\_locks}$ and $\overline{g}_{required\_rq\_locks}$). It only obtains request queue locks not already claimed by p (see $\overline{g}_{missing\_rq\_locks}$). In general, call stack locks cannot be obtained and must be retrieved through lock passing.

Processor p can be assured that each processor, whose request queue lock p obtained, holds its call stack lock. Assume g is one of these processors and assume g does not hold its call stack lock. Processor g must have passed its locks and thus be waiting for the locks to return. Another processor must have obtained g's request queue lock and added g's current feature request. Therefore, g's request queue must still be locked because g is still processing the feature request. This means p could not have obtained g's request queue lock, contradicting the assumption.

Once p obtained all locks and found the precondition to be satisfied, it executes f' body (see `execute_body`). Processor p then checks f's postcondition along with the invariant (see `check_post_and_inv`). Finally, p releases the obtained request queue locks (see `issue` and `pop_obtained_locks`) and returns (see `return`) using channel a to synchronize with the client q.

APPLY FEATURE (NON-ONCE ROUTINE OR FRESH ONCE ROUTINE)

$$
\frac{
\begin{array}{l}
f \in \mathit{ROUTINE} \land (f.\mathit{is\_once} \Rightarrow \sigma.\mathit{fresh}(p, f.\mathit{id})) \\
\sigma.\mathit{handler}(r_0) = p \\
\sigma' \overset{def}{=} \begin{cases} \sigma.\mathit{set\_once\_func\_not\_fresh}(p, f, \mathit{void}) & \text{if } f \in \mathit{FUNCTION} \land f.\mathit{is\_once} \\ \sigma.\mathit{set\_once\_proc\_not\_fresh}(p, f) & \text{if } f \in \mathit{PROCEDURE} \land f.\mathit{is\_once} \\ \sigma & \text{otherwise} \end{cases} \\
\sigma'' \overset{def}{=} \sigma'.\mathit{pass\_locks}(q, p, (\bar{l}_r, \bar{l}_c)).\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n)) \\
\overline{g}_{\mathit{required\_rq\_and\_cs\_locks}} \overset{def}{=} \{p\} \cup \\
\quad \{x \in \mathit{PROC} \mid \exists i \in \{1, \ldots, n\}, g, c \colon \Gamma \vdash f.\mathit{formals}(i) : (!, g, c) \land \\
\quad \sigma''.\mathit{ref\_obj}(r_i).\mathit{class\_type}.\mathit{is\_ref} \land x = \sigma''.\mathit{handler}(r_i)\} \\
\overline{g}_{\mathit{required\_cs\_locks}} \overset{def}{=} \\
\quad \{x \in \overline{g}_{\mathit{required\_rq\_and\_cs\_locks}} \mid x = p \lor \\
\quad (x \neq p \land (\sigma''.\mathit{rq\_locks}(x).\mathit{has}(p) \lor \sigma''.\mathit{cs\_locks}(x).\mathit{has}(p)))\} \\
\overline{g}_{\mathit{required\_rq\_locks}} \overset{def}{=} \overline{g}_{\mathit{required\_rq\_and\_cs\_locks}} \setminus \overline{g}_{\mathit{required\_cs\_locks}} \\
\{g_1, \ldots, g_m\} \overset{def}{=} \overline{g}_{\mathit{missing\_rq\_locks}} \overset{def}{=} \{x \in \overline{g}_{\mathit{required\_rq\_locks}} \mid \neg\sigma''.\mathit{rq\_locks}(p).\mathit{has}(x)\} \\
a' \text{ is fresh}
\end{array}
}{
\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow \\
\quad \langle p :: \texttt{check\_pre\_and\_lock}(f, \overline{g}_{\mathit{missing\_rq\_locks}}); \\
\qquad \texttt{execute\_body}(f, \overline{g}_{\mathit{missing\_rq\_locks}}); \\
\qquad \texttt{check\_post\_and\_inv}(f); \\
\qquad \texttt{issue}(g_1, \texttt{unlock\_rq}); \ldots; \texttt{issue}(g_m, \texttt{unlock\_rq}); \\
\qquad \texttt{pop\_obtained\_locks}; \\
\qquad \texttt{provided } f \in \mathit{FUNCTION} \texttt{ then} \\
\qquad\quad \texttt{read}(\mathit{result}, a'); \texttt{return}(a, a'.\mathit{data}, q) \\
\qquad \texttt{else} \\
\qquad\quad \texttt{return}(a, q) \\
\qquad \texttt{end}; \\
\qquad s_p, \sigma''\rangle
\end{array}
}
$$

Figure 3.46: Non-once or fresh once routine application transition rule

Figure 3.47 describes the `check_pre_and_lock` operation in detail. The operation takes a feature f, whose precondition must be satisfied, and a processor set $\{g_1, \ldots, g_m\}$, whose request queues must be locked. Processor p goes through a number of iterations; in each iteration, it obtains the request queue locks and then evaluates the precondition. If the precondition is not satisfied, then p releases the request queue locks and starts another iteration; otherwise it moves on.

CHECK PRECONDITION AND LOCK

```
                          a is fresh
───────────────────────────────────────────────────────────────
Γ ⊢⟨p :: check_pre_and_lock(f, {g_1, ..., g_m}); s_p, σ⟩ →
   ⟨p :: lock({g_1, ..., g_m});
        provided f.has_pre then
           eval(a, f.pre);
           wait(a);
           provided a.data then
              nop
           else
              issue(g_1, unlock_rq);
              ...
              issue(g_m, unlock_rq);
              pop_obtained_locks;
              check_pre_and_lock(f, {g_1, ..., g_m})
           end
        else
           nop
        end;
        s_p, σ⟩
```

Figure 3.47: Check precondition and lock transition rule

Figure 3.48 describes the execution of the feature body. For once functions, p updates the once status whenever it writes to the result entity by executing the `set_not_fresh` operation.

EXECUTE BODY

$$\frac{}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{execute\_body}(f); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ f \in \mathit{FUNCTION} \wedge f.\mathit{is\_once}\ \texttt{then} \\ \qquad\qquad f.\mathit{body} \\ \qquad\qquad\quad [\mathit{result} := y;\ \texttt{set\_not\_fresh}(f) / \mathit{result} := y] \\ \qquad\qquad\quad [\textbf{create}\ \mathit{result}.y;\ \texttt{set\_not\_fresh}(f) / \textbf{create}\ \mathit{result}.y] \\ \qquad\quad \texttt{else} \\ \qquad\qquad f.\mathit{body} \\ \qquad\quad \texttt{end}; \\ \qquad\quad s_p, \sigma\rangle\end{array}}$$

SET NOT FRESH

$$\frac{\begin{array}{l} f \in \mathit{FUNCTION} \wedge f.\mathit{is\_once} \\ \sigma.\mathit{envs}(p).\mathit{top}.\mathit{names}.\mathit{has}(\mathit{result}.\mathit{name}) \\ \sigma' \overset{def}{=} \sigma.\mathit{set\_once\_func\_not\_fresh}(p, f, \sigma.\mathit{val}(p, \mathit{result}.\mathit{name})) \end{array}}{\Gamma \vdash \langle p :: \texttt{set\_not\_fresh}(f); s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma\rangle}$$

Figure 3.48: Execute body transition rules

For the postcondition evaluation, the informal description allows p to ask one of its suppliers to perform the evaluation on its behalf, i.e., to dispatch the query calls and wait for the results. While this relaxation can improve performance, it only makes sense if f's postcondition is satisfied. If the postcondition is not satisfied, p must throw an exception and handle it in f's context. Hence, p cannot delegate the evaluation. Figure 3.49 reflects this clarification.

Clarification 3.7 (Postconditions). The postcondition of a feature f must be evaluated by the processor executing f. □

To return, processor p removes its top environment and returns any passed locks to q. In case p returns an expanded object and $q \neq p$, p imports the result to q; otherwise p just returns the result r_r. For a query, p posts the result on channel a and notifies q of any returned locks. For a command with lock passing, p uses a to notify q of the returned locks. Figure 3.50 describes these steps.

Check postcondition and invariant

$$\frac{\begin{array}{c} a_{inv} \text{ is fresh} \\ a_{post} \text{ is fresh} \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{check\_post\_and\_inv}(f);\, s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ f.has\_post\ \texttt{then} \\ \qquad\quad \texttt{eval}(a_{post}, f.post);\, \texttt{wait}(a_{post}) \\ \qquad \texttt{else} \\ \qquad\quad \texttt{nop} \\ \qquad \texttt{end}; \\ \qquad \texttt{provided}\ f.class\_type.has\_inv \wedge f.is\_exported\ \texttt{then} \\ \qquad\quad \texttt{eval}(a_{inv}, f.class\_type.inv);\, \texttt{wait}(a_{inv}) \\ \qquad \texttt{else} \\ \qquad\quad \texttt{nop} \\ \qquad \texttt{end}; \\ \qquad s_p, \sigma\rangle \end{array}}$$

Figure 3.49: Check postcondition and invariant transition rule

Return (command)

$$\frac{\sigma' \overset{def}{=} \sigma.pop\_env(p).revoke\_locks(q, p)}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{return}(a, q);\, s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ \sigma.passed(q)\ \texttt{then}\ \texttt{notify}(a)\ \texttt{else}\ \texttt{nop}\ \texttt{end};\, s_p, \sigma'\rangle \end{array}}$$

Return (query)

$$\frac{\begin{array}{l} (\sigma', r_r') \overset{def}{=} \begin{cases} \text{if } r_r \neq void \wedge \sigma.ref\_obj(r_r).class\_type.is\_exp \wedge \sigma.handler(r_r) \neq q \\ \quad (\sigma^*, \sigma^*.last\_imported\_obj) \\ \quad\ \ \textbf{where} \\ \qquad \sigma^* \overset{def}{=} \sigma.import(q, r_r) \\ \text{otherwise} \\ \quad (\sigma, r_r) \end{cases} \\ \sigma'' \overset{def}{=} \sigma'.pop\_env(p).revoke\_locks(q, p) \end{array}}{\Gamma \vdash \langle p :: \texttt{return}(a, r_r, q);\, s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a, r_r');\, s_p, \sigma''\rangle}$$

Figure 3.50: Return transition rules

Figure 3.51 and Figure 3.52 show the transition rules for once routines that are not fresh and attributes. For once functions, processor p gets the once result from the state and returns it. For once procedures, p does not change anything. For attributes, p evaluates the attribute expression and returns the result of the evaluation. Since f, an instance of *ATTRIBUTE*, is also an instance of *EXPRESSION*, p can use f in `eval`.

Apply feature (not fresh once routine)

$$\frac{\begin{array}{l} f \in \mathit{ROUTINE} \wedge f.\mathit{is\_once} \wedge \neg\sigma.\mathit{fresh}(p, f.\mathit{id}) \\ \sigma.\mathit{handler}(r_0) = p \\ \sigma' \overset{def}{=} \sigma.\mathit{pass\_locks}(q, p, (\bar{l}_r, \bar{l}_c)).\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n)) \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \mathtt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \mathtt{provided}\ f \in \mathit{FUNCTION}\ \mathtt{then} \\ \qquad\qquad \mathtt{return}(a, \sigma'.\mathit{once\_result}(p, f.\mathit{id}), q) \\ \qquad\quad \mathtt{else} \\ \qquad\qquad \mathtt{return}(a, q) \\ \qquad\quad \mathtt{end}; \\ \qquad\quad s_p, \sigma'\rangle \end{array}}$$

Figure 3.51: Not fresh once routine application transition rule

Apply feature (attribute)

$$\frac{\begin{array}{l} f \in \mathit{ATTRIBUTE} \\ \sigma.\mathit{handler}(r_0) = p \\ \sigma' \overset{def}{=} \sigma.\mathit{pass\_locks}(q, p, (\bar{l}_r, \bar{l}_c)).\mathit{push\_env\_with\_feature}(p, f, r_0, ()) \\ a' \text{ is fresh} \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \mathtt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \mathtt{eval}(a', f); \\ \qquad\quad \mathtt{wait}(a'); \\ \qquad\quad \mathtt{return}(a, a'.\mathit{data}, q); \\ \qquad\quad s_p, \sigma'\rangle \end{array}}$$

Figure 3.52: Attribute application transition rule

3.3.12 Creation instructions

The semantics of a creation instruction **create** $b.f(e_1, \dots, e_n)$ with target entity b depends on b's type. The following cases exist:

- The entity b has a separate type.
- The entity b has an explicit processor specification and the specified processor already exists.
- The entity b has an explicit processor specification and the specified processor does not yet exist.
- The entity b has a non-separate type.

Figure 3.53 shows the transition rule for the first case. Processor p first creates a new processor q with a new object o referenced by r. It then obtains a request queue lock on q so that it can issue statements. Next, it updates the target entity b with r and calls the creation procedure f on it. Finally, p releases the obtained request queue lock. The invariant for f must hold after the creation. Since every creation procedure is assumed to be exported (see Chapter 2), the invariant gets checked during f's application (see Section 3.3.11).

CREATION INSTRUCTION (SEPARATE)

$$\frac{\begin{array}{l}(d, g, c) \overset{def}{=} \mathit{type\_of}(\Gamma, b) \\ g = \top \\ q \overset{def}{=} \sigma.\mathit{new\_proc} \\ o \overset{def}{=} \sigma.\mathit{new\_obj}(c) \\ \sigma' \overset{def}{=} \sigma.\mathit{add\_proc}(q).\mathit{add\_obj}(q, o) \\ r \overset{def}{=} \sigma'.\mathit{ref}(o)\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1, \dots, e_n); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{lock}(\{q\}); \\ \qquad \texttt{write}(b, r, \mathit{false}); \\ \qquad b.f(e_1, \dots, e_n); \\ \qquad \texttt{issue}(q, \texttt{unlock\_rq}); \\ \qquad \texttt{pop\_obtained\_locks}; \\ \qquad s_p \mid q :: \texttt{nop}, \sigma'\rangle\end{array}}$$

Figure 3.53: Separate creation instruction transition rule

Figure 3.55 and Figure 3.56 cover the two cases where b has an explicit processor specification: Figure 3.55 for the case the specified processor exists and Figure 3.56 for the case it does not. An unqualified explicit processor specification $< x >$ is based on a processor attribute x with an attached type; the specified processor is the one stored in x. A qualified explicit processor specification $< y.handler >$ is based on a non-writable entity y of attached type; the specified processor is the one handling the object referenced by y. Unlike an unqualified specification, a qualified one always specifies an existing processor because it is based on an existing object. Hence, the condition in Figure 3.56 does not cover qualified specifications.

In case the specified processor q exists, processor p creates a new object o with reference r on q. Processor p then locks q's request queue in case it requires that lock for the creation call, i.e., the creation call is a separate call (no callback), and has not already claimed the lock. Processor p then continues as in the separate case. Finally, p release q's request queue lock in case it has obtained it earlier. In case the specified processor does not exist yet, p creates a new processor q with a new object o referenced by r. It then stores q in the processor entity x, locks q's request queue, and updates the target entity b with r. Finally, it calls the creation routine before it releases q's request queue lock.

For the case where b is non-separate, p creates the object on itself, updates the target entity, and executes the creation routine. Figure 3.54 shows this case in detail. The third argument to `write` ensures that p does not copy the newly created object in case it is expanded.

Creation instruction (non-separate)

$$\frac{\begin{array}{l}(d,g,c) \overset{def}{=} type\_of(\Gamma, b)\\ g = \bullet\\ o \overset{def}{=} \sigma.new\_obj(c)\\ \sigma' \overset{def}{=} \sigma.add\_obj(p,o)\\ r \overset{def}{=} \sigma'.ref(o)\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1,\dots,e_n); s_p, \sigma\rangle \rightarrow\\ \quad \langle p :: \texttt{write}(b, r, false);\\ \qquad b.f(e_1,\dots,e_n);\\ \qquad s_p, \sigma'\rangle\end{array}}$$

Figure 3.54: Non-separate creation instruction transition rule

Creation instruction (existing explicitly specified processor)

$$(d, g, c) \overset{def}{=} type\_of(\Gamma, b)$$
$$(\exists x \in ENTITY: g =< x >) \vee (\exists x \in ENTITY: g =< x.handler >)$$
$$q \overset{def}{=} \begin{cases} \sigma.val(p, x.name) & \text{if } \exists x \in ENTITY: g =< x > \\ \sigma.handler(\sigma.val(p, x.name)) & \text{if } \exists x \in ENTITY: g =< x.handler > \end{cases}$$
$$\sigma.procs.has(q)$$
$$o \overset{def}{=} \sigma.new\_obj(c)$$
$$\sigma' \overset{def}{=} \sigma.add\_obj(q, o)$$
$$r \overset{def}{=} \sigma'.ref(o)$$
$$w \overset{def}{=} q \neq p \wedge \neg(q \neq p \wedge (\sigma'.rq\_locks(q).has(p) \vee \sigma.cs\_locks(q).has(p))) \wedge \neg\sigma'.rq\_locks(p).has(q)$$

```
Γ ⊢⟨p :: create b.f(e_1, ..., e_n); s_p, σ⟩ →
   ⟨p :: provided w then
            lock({q})
         else
            nop
         end;
         write(b, r, false);
         b.f(e_1, ..., e_n);
         provided w then
            issue(q, unlock_rq);
            pop_obtained_locks
         else
            nop
         end;
         s_p, σ'⟩
```

Figure 3.55: Existing explicit processor creation instruction transition rule

Creation instruction (non-existing explicitly specified processor)

$$\frac{\begin{array}{l}(d,g,c) \overset{def}{=} type\_of(\Gamma, b)\\ \exists x \in ENTITY: g =< x >\\ < x > \overset{def}{=} g\\ \neg\sigma.procs.has(\sigma.val(p, x.name))\\ q \overset{def}{=} \sigma.new\_proc\\ o \overset{def}{=} \sigma.new\_obj(c)\\ \sigma' \overset{def}{=} \sigma.add\_proc(q).add\_obj(q, o)\\ r \overset{def}{=} \sigma'.ref(o)\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1, \ldots, e_n); s_p, \sigma\rangle \rightarrow\\ \quad \langle p :: \texttt{write}(x, q, false);\\ \qquad \texttt{lock}(\{q\});\\ \qquad \texttt{write}(b, r, false);\\ \qquad b.f(e_1, \ldots, e_n);\\ \qquad \texttt{issue}(q, \texttt{unlock\_rq});\\ \qquad \texttt{pop\_obtained\_locks};\\ \qquad s_p \mid q :: \texttt{nop}, \sigma'\rangle\end{array}}$$

Figure 3.56: Non-existing explicit processor creation instruction transition rule

3.3.13 Flow control instructions

To execute an **if** e **then** s_t **else** s_f **end** instruction or a **if** e **then** s_t **end** instruction, processor p first evaluates the expression e, as shown in Figure 3.57. It then uses the `provided` operation to execute the right branch. To execute a **until** e **loop** s_l **end** instruction, p also first evaluates e. If e is true, p executes the statements s_l, followed by another loop iteration; if e is false, p is done.

3.3.14 Assignment instructions

To assign the value of an expression e to an entity b, processor p evaluates the expression e, waits for the result, and then stores the result into b. Figure 3.58 shows this process. The third argument in `write` ensures that p copies an expanded object before assigning it to b. This copy semantics is not only reflected in the assignment instruction: the `return` operation (see Section 3.3.11) and the `call` operation (see Section 3.3.10) also copy or import expanded objects.

If instruction (with else)

$$\frac{a \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{if}\ e\ \textbf{then}\ s_t\ \textbf{else}\ s_f\ \textbf{end}; s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a, e); \\ \qquad \texttt{wait}(a); \\ \qquad \texttt{provided}\ a.data\ \texttt{then}\ s_t\ \texttt{else}\ s_f\ \texttt{end}; \\ \qquad s_p, \sigma\rangle\end{array}}$$

If instruction (without else)

$$\frac{a \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{if}\ e\ \textbf{then}\ s_t\ \textbf{end}; s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a, e); \\ \qquad \texttt{wait}(a); \\ \qquad \texttt{provided}\ a.data\ \texttt{then}\ s_t\ \texttt{else}\ \texttt{nop}\ \texttt{end}; \\ \qquad s_p, \sigma\rangle\end{array}}$$

Loop instruction

$$\frac{a \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{until}\ e\ \textbf{loop}\ s_l\ \textbf{end}; s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a, e); \\ \qquad \texttt{wait}(a); \\ \qquad \texttt{provided}\ a.data\ \texttt{then}\ \texttt{nop}\ \texttt{else}\ s_l; \textbf{until}\ e\ \textbf{loop}\ s_l\ \textbf{end}\ \texttt{end}; \\ \qquad s_p, \sigma\rangle\end{array}}$$

Figure 3.57: Flow control instructions transition rules

Assignment instruction

$$\frac{a \text{ is fresh}}{\Gamma \vdash \langle p :: b := e; s_p, \sigma\rangle \rightarrow \langle p :: \texttt{eval}(a, e); \texttt{wait}(a); \texttt{write}(b, a.data, true); s_p, \sigma\rangle}$$

Figure 3.58: Assignment instructions transition rule

3.3.15 Termination or blockage

Once the system reaches a configuration where all action queues are empty, i.e., there is no more work to do, then the system terminates. If the system gets stuck in a non-empty configuration, e.g., a deadlock, then it blocks in that configuration.

4 Executable specification

Executable formal specifications support the execution of a program without compiler or runtime support. A tool executes the program according to the executable formal specification. An executable formal specification of a concurrency model is useful in analyzing and verifying the model since test programs can be run and their behavior can be validated. Several frameworks support the development of executable formal specifications, as summarized in Section 4.5. We chose Maude [53, 54] because of its expressiveness, performance, and rich execution capabilities. This chapter presents the main techniques in mapping SCOOP's formal specification to Maude, namely mapping ADTs and transition rules. This chapter also discusses how to use Maude's multifaceted execution capabilities to run test programs. The techniques presented in this chapter apply to the formal specification in Chapter 3 as well as the formal specifications for the asynchronous exception mechanism in Chapter 6 and the data sharing mechanism in Chapter 7. In addition to these techniques, this chapter presents background material on Maude for readers who are not yet familiar with this framework. Readers who are already familiar with Maude can jump directly to Section 4.2.

4.1 Maude

Maude [53, 54] offers a programming language for theories in *membership equational logic* [32,145] and *rewriting logic* [38,144]. A membership equational logic is a Horn logic whose basic predicates either express membership assertions, stating that a *term* built with *operators* belongs to a certain *sort*, or equations between terms, stating that two terms are equal under a certain condition. Some operators are *constructors*. Terms only built with constructors cannot be further reduced using equations. A sort can be a *subsort* of another sort. A rewrite logic has *rewrite rules* that express conditional rewrites between terms. Maude combines the two by rewriting terms according to the rewrite logic modulo the membership equational logic. It uses the equations to reduce a term to a canonical form based on constructors, applies fitting rewrite rules, and then once again reduces the resulting term using the equations.

A *functional module* (**fmod** keyword) contains a membership equational theory. It defines the sorts (**sort** keyword), subsort relationships (**subsort** keyword), operators (keywords **op** and **ctor** for constructors), and equations (keywords **eq** and **ceq**) for the theory. The module can import a submodule in three modes; these modes do not have an operational difference but simply express different import promises. In *protecting mode* (**protecting** keyword), the module does not extend any of the submodule's sorts with new constructors, and it does not introduce any equivalences between imported operators of these sorts. In *extending mode* (**extending** keyword), the module can extend the sorts, but does not introduce any equivalences between imported operators. In *including mode* (**including** keyword), the module can do both, which is always a safe choice. The following functional module, adapted from the Maude literature [53], defines a theory for natural numbers:

```
fmod NAT is
  protecting BOOL .
  sort Nat .
  sort Int .
  subsort Nat < Int .

  var n m : Nat .

  op 0 : −> Nat [ctor] .
  op s(_) : Nat −> Nat [ctor] .

  op _ < _ : Nat Nat −> Bool .
  eq s(n) < s(m) = n < m .
  eq 0 < s(n) = true .
  eq s(n) < 0 = false .
  eq 0 < 0 = false .

  op min(_, _) : Nat Nat −> Nat .
  ceq min(n, m) = n if n < m .
  ceq min(n, m) = m if m < n .
  ceq min(n, m) = n if n = m .
endfm
```

First, the module imports another module for boolean values. It then defines a sort for natural numbers as a subsort of integers. It also defines some variables to be used throughout the module. A variable can either be declared globally as shown above or inline as in *n*:*Nat*. The module declares several operators using underscores to represent arguments. Two operators define the structure of natural numbers using Peano's successor concept. These operators are constructors, as indicated by the **ctor** attribute. Other possible attributes are **assoc** for associative operators, **comm** for commutative operators, and **id** for operators with an identity element. These attributes serve as implicit equations. The next operator defines the structure of a comparison. In the module, four equations reduce a comparison to a boolean value. The final operators defines the minimum of two natural numbers with three conditional equations.

A *system module* (**mod** keyword) contains a rewrite theory with unconditional rewrite rules (**rl** keyword) and conditional rewrite rules (**crl** keyword). Rewrite rules can have attributes. For example, the attribute **owise** makes a rule applicable only if no other rule applies. The system module can also contain a membership equational theory. As with functional modules, the system module can import other modules in three modes. Importing a functional module works as before whereas importing a system module in protecting or extending mode additionally promises that the reachability relation between imported operators remains unchanged. The following system module defines a buffer with natural numbers:

```
mod BUFFER is
  protecting NAT .
  protecting LIST{Nat} .

  var l l' : List{Nat} .
  var n : Nat .

  sort Buffer .

  op buffer(_) : List{Nat} -> Buffer [ctor] .

  rl [produce] : buffer(l) => buffer(l .append(0 ;)) .
  crl [consume] : buffer(l) => buffer(l')
    if not l .isEmpty ∧ l' := l .pop .
endm
```

This system module defines an unconditional rule (see **rl**) to put a natural number into a buffer. It also defines a conditional rule (see **crl**) to consume a number, provided the buffer is not empty. To define the buffer, the system module includes the parametrized *LIST* module – a custom module with operators that follow the object-oriented dot notation.

A module can be *parametrized*. To use such a module, each parameter must be bound to another module, i.e., the *parameter module*. A *theory* defines the sorts, subsort relationships, operators, equations, and rewrite rules that the parameter module must provide. A *view* (**view** keyword) specifies how the parameter module satisfies that theory. For instance, the parameter of the list module represents the theory of a list element. The parameter has the built-in trivial theory *TRIV* associated to it, and the theory has only one restriction on a module: it must have a sort. The following view therefore specifies that *NAT* satisfies *TRIV* by mapping the sort *Nat* to the theory's sort *Elt*:

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

4.2 Abstract data types

Two functional modules represent one ADT. The *ADT sort module* defines a sort to represent the ADT along with subsort relationships. The *ADT module* then includes the ADT sort module to implement the ADT features with operators and equations. The two functional modules are necessary so that the ADT module itself can use instances of the ADT in parametrized collections. A single ADT module would create a circular dependency between the ADT module and the collection module. If instances of the ADT can become members of a parametrized collection, then the ADT has a view to facilitate this. For instance, *REGIONS* from Section 3.2 leads to the following two functional modules; a view is not necessary for *REGIONS*:

```
fmod REGIONS−SORTS is
  sort Regions .

endfm
fmod REGIONS is
  including REGIONS−SORTS .
  ...
endfm
```

The ADT module includes all the ADT modules and collection modules on which it depends. For the *REGIONS* ADT, the following includes are necessary:

```
including REGIONS−SORTS .
including BOOL .
including REF .
including PROC .
including MAP{Proc, RefSet} .
including MAP{Proc, Bool} .
including MAP{Proc, Proc} .
including MAP{Proc, ProcSetList} .
including MAP{Proc, RefSetList} .
including PAIR{ProcSet, ProcSet} .
including SET{ProcSet} .
```

The ADT module of an ADT defines one constructor that determines the internal structure of an ADT instance. This structure reflects the data of an ADT instance as defined by the queries. A number of variables relate to an ADT instance and its data. For *REGIONS*, the internal structure holds the handled objects *ho* as a map *Map*{*Proc*, *RefSet*} and the last added processor *lap*. The map also keeps the available processors in the key set of the map. Next, the internal structure remembers which request queues and call stacks are locked using two maps *rql* and *csl* of type *Map*{*Proc*, *Bool*}. The next four items *orq*, *ocs*, *rrq*, and *rcs* manage the obtained and retrieved locks. Finally, *lp* remembers which processors passed locks.

```
op regions : Map{Proc, RefSet} Proc Map{Proc, Bool} Map{Proc, Bool} Map{Proc,
    ProcSetList} Map{Proc, Proc} Map{Proc, ProcSetList} Map{Proc, ProcSetList} Map
    {Proc, Bool} −> Regions [ctor] .

var k : Regions .
var p : Proc .
var r : Ref .
var ho : Map{Proc, RefSet} .
var lap : Proc .
var rql csl lp : Map{Proc, Bool} .
var orq rrq rcs : Map{Proc, ProcSetList} .
var ocs : Map{Proc, Proc} .
```

The ADT module then defines operators and equations for the ADT's features. Each query leads to one operator and one equation. The operator reflects the structure of the query: the syntax along with the sorts for the ADT instance to operate on, the formal arguments, and the result. The equation links the query to the internal structure of the ADT instance. If the query has a precondition, then the equation has a corresponding condition. The following code shows the first three queries of *REGIONS*; for space reasons, precedence values and formatting specifications are omitted:

```
op _.procs : Regions -> Set{Proc} .
eq regions(ho, lap, rql, csl, orq, ocs, rrq, rcs, lp) .procs = ho .keys .

op _.handledObjs(_) : Regions Proc -> Set{Ref} .
ceq regions(ho, lap, rql, csl, orq, ocs, rrq, rcs, lp) .handledObjs (p) = ho[p] .values
  if ho .keys .has(p) .

op _.rqLocked(_) : Regions Proc -> Bool .
ceq regions(ho, lap, rql, csl, orq, ocs, rrq, rcs, lp) .rqLocked (p) = rql[p] .
  if ho .keys .has(p) .
```

Each command also leads to one operator and one equation. However, the structure of a command is different because the result of a command is an updated ADT instance. The equation reflects the axioms of the command: it defines how to rewrite a command call on an ADT instance into an updated ADT instance. The equation can define auxiliary terms in the condition. As before, the condition also contains the precondition of the command. The following code shows the two commands to add processors and objects; in there, the auxiliary operator *refIdUnique* returns whether the regions already contain a given reference or not.

```
op _.addProc(_) : Regions Proc -> Regions .
ceq k .addProc (p) =
  regions(
    (ho .insert(p --> empty)),
    p,
    (rql[p] ::= false),
    (csl[p] ::= true),
    (orq[p] ::= nil),
    (ocs[p] ::= p),
    (rrq[p] ::= nil),
    (rcs[p] ::= nil),
    (lp[p] ::= false)
  )
  if
    regions(ho, lap, rql, csl, orq, ocs, rrq, rcs, lp) := k ∧
    not k .procs .has(p) .

op _.addObj(_, _) : Regions Proc Obj -> Regions .
ceq k .addObj (p, r) = regions((ho .insert(p --> (ho[p] U {r}))), lap, rql, csl, orq, ocs,
    rrq, rcs, lp)
  if
    regions(ho, lap, rql, csl, orq, ocs, rrq, rcs, lp) := k ∧
    k .procs .has(p) ∧ k .refIdUnique(r) .
```

A constructor is similar to a command with the difference that it does not operate on an ADT instance. The constructor of *REGIONS* initializes the internal structure without any processors; *noProc* denotes no processor:

```
op new REGIONS.make : -> Regions .
op noProc : -> Proc [ctor] .
eq new REGIONS.make = regions(empty, noProc, empty, empty, empty, empty, empty,
    empty, empty) .
```

With the ADT modules, Maude can reduce feature call chains. These feature call chains can be built because constructors and commands always return a new or updated ADT instance that can then be used by a subsequent command or query call to operate on.

4.3 Transition rules

To represent transition rules in Maude, some basic support is necessary. A number of functional modules model a SCOOP program using sorts, operators, and equations for programs, settings, classes, features, expressions, instructions, and types. Other functional modules model operations. Instructions and operations are subsorts of statements. Views ensure that they can be stored in parametrized collections. A new system module defines sorts and operators for action queues and configurations. A list of action queues is partitioned by the associative and commutative parallel operator.

```
sorts Configuration ActionQueue ActionQueueList .
subsort ActionQueue < ActionQueueList .

op _ :: _ : Proc List{Statement} -> ActionQueue [ctor] .
op _ | _ : ActionQueueList ActionQueueList -> ActionQueueList [ctor assoc comm] .
op _ |- _ , _ , _ : Program ActionQueueList Nat State -> Configuration [ctor] .
```

By comparing this implementation to the configuration in Section 3.3, one can notice two deviations. First, the configuration has an additional natural number to count the number of steps in the execution. This counter provides a continuous stream of numbers and is used to create fresh identifiers. This link between the step numbers and the identifiers helps in debugging because one can easily determine in which step Maude created an identifier. Second, each configuration has a program associated with it. This program represents the typing environment. In Section 3.3, the typing environment only occurs as part of the transition definition. With this deviation, Maude does not have to add the typing environment to each configuration before executing a transition rule. With this basic support, the system module can implement transition rules. First, it defines variables:

```
var p q : Proc .
var qs : Set{Proc} .
var σ : State .
var f : Feature .
var sw sp sq : List{Statement} .
var a : Channel .
var ic : Nat .
```

Each SCOOP operation leads to a sort, a subsort relationship, and an operator. The following code represents the issue operation from Section 3.3; as before, precedence values and formatting specifications are omitted:

```
sort Issue .
subsort Issue < Operation .

op issue(_, _) : Proc List{Statement} −> Issue [ctor] .
```

Each transition rule leads to a conditional rewrite rule. The condition of the rewrite rule contains the premise of the transition rule. The label contains the name of the transition rule. The rewrite arrow => separates the start configuration from the result configuration. For instance, the rule for `issue` becomes:

```
crl [issueSeparateCallback] :
  (Γ |− p :: issue(q, sw) ; sp | q :: sq, ic, σ) =>
  (Γ |− p :: sp | q :: sw sq, ic + 1, σ)
  if
    q =/= p and (σ .rqLocks(q).has(p) or σ .csLocks(q).has(p)) ∧
    not σ .passed(p) and σ .csLocks(p).has(q)
```

4.4 Executing programs

To execute a program, Maude expects an initial configuration that captures the starting point of the program. The initial configuration can be given as a list of classes with settings denoting the root class and the root procedure. A rewrite rule transforms this construct into the initial configuration (see Section 3.3.2). Maude offers a number of facilities, using different strategies, to execute programs under different schedules.

4.4.1 Rewrite and frewrite

The **rew** command takes an initial configuration and an upper bound for the number of transition rules. It applies fitting transition rules using a rule-fair top-down strategy until it reaches the upper bound, or it finds no more fitting transition rules. The **frew** command is similar, but it uses a rule- and position-fair strategy. For example, the following invocation applies 100 transition rules to execute '*make*:

```
rew [100]
  (
    (
      class 'A create {'make}
        (
          procedure {'ANY} 'make (nil)
            require
              True
            local
              nil
            do
              ...
            ensure
              True
            rescue
              nil
          end ;
        )
      end ;
    )
    settings ('A, 'make)
  ) .
```

The **rew** command becomes problematic when a sequence of transitions results in action queues that are equal to the action queues at the beginning. In such a case, Maude applies the same transition rules using the same action queues over and over again instead of continuing differently. Priorities in the action queues solve this issue. Maude brings configurations into a canonical form before applying transition rules. During this reduction, Maude orders the action queues since the parallel operator | is commutative and associative. The priorities influence how Maude orders the action queues; consequently, the priorities influence which action queues Maude uses next. The following code shows the changes. An action queue has an additional natural number to associate a priority to the processor; *yield* decreases the priority by increasing the number:

```
op {_} _ :: _ : Nat Proc List{Statement} -> ActionQueue [ctor] .

sort Yield .
subsort Yield < Operation .

op yield : -> Yield [ctor] .
rl [yield] :
  (Γ |- {i} p :: yield ; sp, ic, σ) => (Γ |- {i + 1} p :: sp, ic + 1, σ) .
```

For example, when a processor evaluates a precondition to false, it tries again (see Section 3.3.11) and hence the resulting action queue remains unchanged. With priorities, the processor can decrease the priority in its action queue after determining that the precondition is not satisfied, causing Maude to focus on another action queue. The following code shows this in more detail:

```
crl [checkPreAndLock] :
  (Γ |- {i} p :: checkPreAndLock(f, qs) ; sp, ic, σ) =>
  (Γ |- {i} p ::
    lock(qs) ;
    provided f .hasPre then
      eval(a, f .pre) ;
      wait(a) ;
      provided a .data then
        nop ;
      else
        nissue(qs, unlockRq ;) *** Issue unlockRq to every processor in qs.
        popObtainedLocks ;
        yield ;
        checkPreAndLock(f, qs) ;
      end ;
    else
      nop ;
    end ;
    sp, ic + 1, σ)
  if a := fresh(ic, 1) . *** Create a fresh channel.
```

4.4.2 Strategies

The **rew** and **frew** commands use built-in strategies. In some cases, one can direct these built-in strategies to prefer one processor over another by inserting blocking instructions in the code. However, in general this approach is not sufficient in enforcing a particular schedule. For this reason, the **srew** command [63] extends Maude with custom strategies. A strategy takes a set of start configurations and produces a set of result configurations that can be reached with the strategy. The **srew** command applies a given strategy to a given initial configuration and returns the result. An important subset of the many strategies the command supports is as follows:

- The strategy *idle* returns the start configurations.
- The strategy *tl* applies the transition rule with label *tl* to the start configurations and returns all possible result configurations. The strategy *tl*{*s1*, ..., *sn*} applies the transition rule with label *tl* and applies substrategy *si* for the i-th transition in the condition of rule *tl*. This strategy is indispensable for systems with more than one processor. Many transition rules apply to only one processor. For a system with more processors, the parallelism rule (see Section 3.3.1) has a transition in its condition that allows one processor to transition in parallel to the other processors. A substrategy can specify which transition rule to use. For example, *parallelism*{*lock*} specifies that a processor should obtain locks in parallel to the other processors.
- The strategy *s1* ; *s2* first applies strategy *s1*. It then applies *s2* on the resulting configuration. The strategy $s*$ applies *s* zero or more times, and $s+$ applies *s* one or more times.
- The strategy *s1* | *s2* either applies *s1* or *s2*. It is useful to allow multiple execution paths such as when multiple transition rules specify one operation with variants, and it is unclear which transition rule will apply. For example, *conditionalTrue* | *conditionalFalse* specifies that the system either evaluates the condition of a `provided` operation to true or false.
- The strategy *match p* adds all start configurations matching pattern *p* to the set of result configurations. This strategy is useful in case multiple processors can take the same transition. As a substrategy of the parallelism rule,

it can specify which processor should transition by only matching configurations with that processor. For example, *parallelism*{(*match* (Γ:*Program* |− *proc*(5) :: *sl*:*List*{*Statement*}, *ic*:*Nat*, σ:*State*)) ; *unlockRq*} specifies that only processor 5 should unlock its request queue.

- The strategy *s1* ? *s2* : *s3* checks whether *s1* produces at least one result configuration, in which case it applies *s2* to the result configurations. Otherwise, it applies *s3* to the start configurations. This strategy combined with *idle* is useful when it is unclear whether a particular transition will be taken or not. For example, *unlockRq* ? *idle* : *idle* specifies that a processor may or may not unlock its request queue.

Using these strategies, one can specify an execution by concatenating one strategy per transition. Each strategy is a parallelism rule strategy with a substrategy consisting of *match* followed by a transition rule strategy. *match* selects the subset of processors that take the transition using the subsequent transition rule strategy. To specify alternative execution paths, one nests concatenations in a | strategy; to specify that a transition may or may not happen, one uses the conditional strategy. For example, the following invocation first gives p_1 a chance to unlock its request queue and then gives a chance to p_2. Here, each processor unlocks only if it can:

```
srew
  (initial configuration)
using
  ... ;
  ((parallelism{
    (match (Γ:Program |− proc(1) :: sl:List{Statement}, ic:Nat, σ:State)) ;
    unlockRq})
  ? idle : idle) ;
  ((parallelism{
    (match (Γ:Program |− proc(2) :: sl:List{Statement}, ic:Nat, σ:State)) ;
    unlockRq})
  ? idle : idle) .
```

4.4.3 Search

None of the commands so far iterates through all possible schedules; the **search** command fills this gap. It uses an exhaustive breadth-first strategy to find executions starting from a given initial configuration and ending in a given terminal configuration. It takes the terminal configuration, an upper bound on desired solutions, and a maximum search depth. The terminal configuration can contain placeholder variables and a property that has to be satisfied. For example, the following invocation looks for a configuration where the request queues of two processors *p* and *q* are locked:

```
var Γ : Program .
var aqs : ActionQueueList .
var p q : Proc .
var ic : Nat .
var sp sq : List{Statement} .
var σ : State .

search
  (initial configuration)
=>*
  (Γ |− p :: sp | q :: sq | aqs, ic, σ)
such that
  σ .rqLocked(p) and σ .rqLocked(q) .
```

The **search** command can be used for bounded or unbounded model checking of an invariant *i*; the invariant holds if **search** reports that no execution ends in a configuration satisfying $\neg i$.

4.4.4 LTL model checker

In addition to **search**, Maude offers a reasonably fast [64] functional module *MODEL−CHECKER* with the *modelCheck* operation. The *modelCheck* operation takes an initial configuration with a LTL [97, 180] property and either returns a counterexample execution or confirms that the property is satisfied. For example, the following invocation checks whether the call stack of p_1 is always locked:

```
mod SCOOP−PREDICATES is
  protecting SYSTEM .
  including SATISFACTION .
  subsort Configuration < MState .

  var Γ : Program .
  var cl : List{Class} .
  var stt : Settings .
  var aqs : ActionQueueList .
  var p : Proc .
  var ic : Nat .
  var sp : List{Statement} .
  var σ : State .

  op csLocked : Proc −> Prop .
  ceq (Γ |− p :: sp | aqs, ic, σ) |= csLocked(p) = false
    if
      not σ .regions .csLocked(p) .
  eq (Γ |− aqs, ic, σ) |= csLocked(p) = true [owise] .
  eq cl stt |= callStackLocked(p) = true .
endm

mod SCOOP−MODEL−CHECKER is
  protecting SCOOP−PREDICATES .
  including MODEL−CHECKER .
  including LTL−SIMPLIFIER .
endm

red modelCheck((initial configuration), [] csLocked(proc(1))) .
```

The model checker has a module *SATISFACTION* with sort *State* for model checker states and sort *Prop* for model checker properties. It also defines an operator **op** _|=_ : *State Prop −> Bool* that evaluates a property on a state. The sort *Configuration* (see Section 4.3) must be a subsort of *State*. Since the executable formal specification also declares a sort *State*, it is necessary to resolve the conflict, e.g., by renaming the sort *State* in *SATISFACTION* and *MODEL−CHECKER* to *MState*. Accordingly, the custom module *SCOOP−PREDICATES* makes the sort *Configuration* from *SYSTEM* a subsort of *MState* and declares the property *csLocked*. *SCOOP−MODEL−CHECKER* defines the context for the *modelCheck* invocation by importing *SCOOP−PREDICATES* along with the main module *MODEL−CHECKER* and the simplification module *LTL−SIMPLIFIER*.

LTL properties assume executions to be infinitely long. For finite executions, the model checker automatically converts any terminal configuration into an infinite sequence that repeats the terminal configuration. Programs with infinite executions can cause the model checker not to terminate. For example, a processor that infinitely often evaluates its precondition to false generates an infinite sequence of differing configurations because the configurations contain a changing instruction counter.

4.5 Other frameworks

A number of frameworks other than Maude support the development of executable formal specifications. CafeOBJ [57] is a sister framework of Maude. Both are descendants of OBJ2 [77]. The K framework [187] builds on top of Maude. Its rewrite rules generalize over traditional ones by being able to specify which parts of a term they read, write, or do not care.

PLT Redex [70] is a language designed for specifying and debugging operational semantics, requiring a grammar and reduction rules as the only input. It can search for all reductions of a term or only for reductions into a specified term, and it can test randomly generated terms.

ELAN [30] uses custom strategies to rewrite a term according to a set of rewrite rules. Similarly, Stratego/XT [33] supports rewrite rules with strategies.

The Relational Meta Language (RML) [74] comes with a tool to generate efficient C interpreters from natural semantics [120].

Several frameworks exist to develop and execute formal specifications using Abstract State Machines (ASM) [29, 85]. Examples include AsmGofer [191] and AsmL [86].

Proof assistants like Isabelle/HOL [165], Coq [22], PVS [175], HOL4 [197], or Twelf [178] can also be used to develop and interpret formal specifications. Some of them are even capable of producing a standalone interpreter. For example, Lochbihler [136] uses Isabelle to generate an interpreter from a formal specification of Java. Ott [193] provides a metalanguage and a tool to express formal specifications and to compile them into code that can be interpreted by proof assistants.

5 Testing

The executable formal specification is useful to execute a program without compiler and runtime support, thus facilitating early testing of the model. Testing the executable formal specification with an appropriate test suite enables the identification of flaws and clarifications in the model. This chapter demonstrates the usefulness of testing on SCOOP using a test suite. It presents the resulting flaws and clarifications.

5.1 Test suite

In testing a concurrency model, it is important to use test programs that focus on individual aspects of the concurrency model as well as test programs that reflect real-world scenarios. The former supports a systematic coverage of the model; the latter enables study of all aspects together. The test suite for SCOOP contains both types of test programs.

Table 5.1 summarizes the test programs for individual aspects. Each test program covers one or more aspects, and each aspect is covered by at least one program. The aspects also include failures and passive processors, as discussed in detail in Chapter 6 and Chapter 7. Since the exception mechanism and passive processors are new developments, several test programs cross-check the new aspects with the existing ones.

The real-world scenarios include parallel searching, share market simulation, pipeline computation, producer-consumer, and concurrent linked list:

- *Parallel searching*. Multiple parallel searchers explore a search space.
- *Share market simulation*. Multiple investors buy and sell shares.
- *Pipeline computation*. Multiple stages in a pipeline process a stream of data.

- *Producer-consumer.* A producer and a consumer exchange data.
- *Concurrent linked list.* Each node of the list is handled by its own processor.

Table 5.1: Overview of aspect tests. The marks indicate the covered aspects.

test	locking, preconditions, and unlocking	wait by necessity	lock passing	separate callbacks	once routines	import	explicit processor specifications	postconditions and invariants	failures	passive processors
1	X		X						X	
2	X									
3		X	X							
4			X							
5			X	X						
6					X					
7						X				
8							X			
9								X	X	
10				X					X	
11					X				X	
12									X	
13									X	
14			X							X
15					X					X
16									X	X

While the test suite covers failures and passive processors, this chapter only discusses flaws and clarifications found for SCOOP without the two new mechanisms. Chapter 6 and Chapter 7 complete the discussion for the new mechanisms.

5.2 Separate callbacks

A separate callback is characterized by the following condition (see Section 3.3.3): at the moment of a feature call, the supplier claims a lock on the client.

5.2.1 Flaw

While the separate callback condition adequately captures simple scenarios with only two processors, it fails to capture more complex scenarios:

```
class A create make feature
  b: separate B
  c: separate C

  make
    do
      create b.make; create c.make
      f (b)
    end

  f (b: separate B)
    do
      b.g (c, Current)
    end

  k: separate D
    do
      create Result.make
    end
end

class B create make feature
  make
    do end

  g (c: separate C; a: separate A)
    do
      c.h (a)
    end
end

class C create make feature
  make
    do end

  h (a: separate A)
    local
      t: separate D
    do
      t := a.k
    end
end

class D create make feature
  make
    do end
end
```

Using Maude's **rew** command, the system creates an object of type A on a new root processor p_1; processor p_1 then creates two objects on two new processors p_3 and p_5 and executes feature f:

p_1 :: orq: ({}, {p_3}) rrq: ({}, {}) rcs: ({}, {}) locked
p_3 :: orq: () rrq: () rcs: () locked
p_5 :: orq: () rrq: () rcs: () unlocked

Processor p_1 then calls processor p_3. The call is synchronous as it involves lock passing triggered by the reference to the current object as an actual argument. Processor p_1 passes both the request queue lock on p_3 and its own call stack lock. Processor p_3 then obtains the request queue lock of p_5:

p_1 :: orq: ({}, {p_3}) rrq: ({}, {}) rcs: ({}, {}) locked passed
p_3 :: orq: ({p_5}) rrq: ({p_3}) rcs: ({p_1}) locked
p_5 :: orq: () rrq: () rcs: () locked

Processor p_3 then passes all its locks to processor p_5 during a synchronous call with lock passing:

p_1 :: orq: ({}, {p_3}) rrq: ({}, {}) rcs: ({}, {}) locked passed
p_3 :: orq: ({p_5}) rrq: ({p_3}) rcs: ({p_1}) locked passed
p_5 :: orq: ({}) rrq: ({p_5, p_3}) rcs: ({p_1, p_3}) locked

Finally, processor p_5 synchronously calls processor p_1, which is waiting for its synchronous call to return. However, the feature call does not qualify as a separate callback because processor p_1 does not claim a lock on processor p_5. Therefore, processor p_5 performs a regular separate feature call and adds its feature request to the end of processor p_1's request queue. At this point, all processors wait. Maude cannot process any of the wait operations because no processor completes its feature request.

5.2.2 Clarification

This design flaw can be resolved by fixing the separate callback condition. Figure 5.1 shows the changes to the issuing mechanism. The separate callback condition also appears in the transition rules for feature calls (see Figure 3.44 and Figure 3.45), in the transition rule for feature applications (see Figure 3.46), and in the transition rule for creating an object (see Figure 3.55). These rules must be adapted accordingly.

ISSUE (SEPARATE)

$$\frac{\begin{array}{l} q \neq p \wedge \neg \sigma.\mathit{passed}(q) \\ \neg \sigma.\mathit{passed}(p) \wedge \sigma.\mathit{rq\_locks}(p).\mathit{has}(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma \rangle \rightarrow \langle p :: s_p \mid q :: s_q; s_w, \sigma \rangle}$$

ISSUE (SEPARATE CALLBACK)

$$\frac{\begin{array}{l} q \neq p \wedge \sigma.\mathit{passed}(q) \wedge \neg \sigma.\mathit{passed}(p) \wedge \sigma.\mathit{cs\_locks}(p).\mathit{has}(q) \\ \neg \sigma.\mathit{passed}(p) \wedge \sigma.\mathit{cs\_locks}(p).\mathit{has}(q) \end{array}}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma \rangle \rightarrow \langle p :: s_p \mid q :: s_w; s_q, \sigma \rangle}$$

Figure 5.1: Separate callback clarification

A separate callback is now characterized as: the supplier has passed its locks, and the client holds the supplier's call stack lock (first condition). This condition implies that the supplier is waiting because it passed its locks over a chain of feature calls all the way to the client. In this situation, it is necessary for the supplier to process a feature request right away; otherwise, a deadlock would occur. In addition to the simple scenarios with only two processors, this condition also captures more complex scenarios such as the one shown here.

Clarification 5.1 (Separate callback). A separate callback is characterized as follows: the supplier has passed its locks, and the client holds the supplier's call stack lock. □

5.3 Separate once functions

A *once function* gets executed at most once in a context. Once functions declared as separate have a once per system semantics. Non-separate once functions have a once per processor semantics. The result from the first execution in a context becomes the result of all future executions in the same context. If a once function has been executed in a context, it is not fresh in that context; otherwise it is fresh.

Two transition rules describe how a processor p processes a feature request for a once routine (see Figure 3.46 and Figure 3.51). The first transition rule only applies to once routines f that are fresh in the context p. Processor p immediately sets f to not fresh with the void result. It then updates the once result during the execution. The second transition rule only applies to once routines f that are not fresh in the context p. In this case, p returns the previous result on p.

5.3.1 Flaw

This specification is problematic when a once function is of separate type and two processors execute the once function concurrently. The following program shows this design flaw in more detail:

```
class A create make feature
  make
    local
      b1, b2: separate B
    do
      create b1.make; create b2.make
      f (b1, b2)
    end

  f (b1: separate B; b2: separate B)
    do
      b1.g
      b2.g
    end
end

class B create make feature
  make
    do end


  g
    local
```

```
      c: separate C
    do
      c := h
    end

  h: separate C
    once
      create Result
    end
end

class C create make feature
  make
    do end
end
```

A new root processor p_1 creates two objects on two new processors p_3 and p_5. It then asks both processors to asynchronously execute g, causing both of them to assign the result of the separate once function h to c. One of them is first and computes the once result; the other uses the existing result. After the assignment, both processors should store a non-void value in c, as expressed for one of the processors in the following formula [] ˜ *voidOnceResult*(*proc*(5)):

```
var Γ : Program .
var cl : List{Class} .
var stt : Settings .
var aqs : ActionQueueList .
var p : Proc .
var ic : Nat .
var sp : List{Statement} .
var σ : State .

op voidOnceResult : Proc −> Prop .
ceq (Γ |− p :: sp | aqs, ic, σ) |= voidOnceResult(p) = true
  if
    σ .rqLocks(p) .isEmpty /\
    not (σ .envs(p) .isEmpty) /\
    σ .envs(p) .top .names .has ('c) /\
    (σ .val(p, 'c)) == void .
eq cl stt |= voidOnceResult (p) = false .

red modelCheck((initial configuration), [] ˜ voidOnceResult(proc(5))) .
```

The property *voidOnceResult* identifies the step after the assignment as the configuration where the processor has released its locks, but still has a variable environment with one variable named 'c. The property is true if the variable stores the void reference. The model checker tries to verify that *voidOnceResult* is globally false for p_5. However, it does find a lengthy counterexample (more than 60'000 lines). Processors p_3 is first and sets h to not fresh with the void result (see Figure 3.46). Right before p_3 creates the result, p_5 determines that h is not fresh and returns the existing result (see Figure 3.51). At that point, p_3 has not created the result yet, and thus p_5 returns the void result.

5.3.2 Clarification

This design flaw can be fixed with a new status sequence for once routines: initially, a once routine is fresh in a context; just after its first execution in the context, it becomes not fresh and *not stable*; finally, it becomes not fresh and *stable* when the first execution is over. The processor that executes the once routine for the first time in a context is the *stabilizer*. With this new status sequence, a processor different from the stabilizer can now be prevented from getting the result too soon, while still allowing the stabilizer to recursively call the once function without blocking.

Figure 5.2 shows the changes to *HEAP* and the `set_not_fresh` operation. One new query returns whether a once routine is stable. Another new query returns the stabilizer. The commands to set a once routine not fresh take a new stability flag; `set_not_fresh` sets this flag to false because it is only used in the middle of a feature application. The state facade (see Section 3.2.6) mirrors these changes.

The transition rule for fresh once routines (see Figure 3.46) must be updated. The stabilizer p sets the once routine f to not fresh and not stable. Upon returning, it sets f to stable by passing a new flag and f to the `return` operation, as shown in Figure 5.3. The condition for not fresh once routines (see Figure 3.51) becomes $\neg\sigma.fresh(p, f.id) \wedge (\sigma.stable(p, f.id) \vee \sigma.stabilizer(p, f.id) = p)$. When a processor applies a once routine that is not fresh, it waits until the routine is stable. Upon returning, the processor does not alter the status of the routine. The condition for attributes remains unchanged (see Figure 3.52), but the invocation of `return` must be adapted.

$stable\colon HEAP \to PROC \to ID \nrightarrow BOOLEAN$
  $h.stable(p, i)$ **require**
    $\neg h.fresh(p, i)$
$stabilizer\colon HEAP \to PROC \to ID \nrightarrow PROC$
  $h.stabilizer(p, i)$ **require**
    $\neg h.fresh(p, i)$
$set\_once\_func\_not\_fresh\colon HEAP \to PROC \to FEATURE \nrightarrow BOOLEAN \to REF$
$\nrightarrow HEAP$
  $h.set\_once\_func\_not\_fresh(p, f, st, r)$ **require**
    $f \in FUNCTION \wedge f.is\_once$
    $r \neq void \Rightarrow h.refs.has(r)$
  axioms
    $(\exists d, c\colon \Gamma \vdash f : (d, \bullet, c)) \Rightarrow$
      $\neg h.set\_once\_func\_not\_fresh(p, f, st, r).fresh(p, f.id) \wedge$
      $h.set\_once\_func\_not\_fresh(p, f, st, r).stable(p, f.id) = st \wedge$
      $h.set\_once\_func\_not\_fresh(p, f, st, r).stabilizer(p, f.id) = p \wedge$
      $h.set\_once\_func\_not\_fresh(p, f, st, r).once\_result(p, f.id) = r$
    $(\exists d, c\colon \Gamma \vdash f : (d, p, c) \wedge p \neq \bullet) \Rightarrow \forall q \in PROC\colon$
      $\neg h.set\_once\_func\_not\_fresh(p, f, st, r).fresh(q, f.id) \wedge$
      $h.set\_once\_func\_not\_fresh(p, f, st, r).stable(q, f.id) = st \wedge$
      $h.set\_once\_func\_not\_fresh(p, f, st, r).stabilizer(q, f.id) = p \wedge$
      $h.set\_once\_func\_not\_fresh(p, f, st, r).once\_result(q, f.id) = r$
$set\_once\_proc\_not\_fresh\colon HEAP \to PROC \to FEATURE \nrightarrow BOOLEAN \to HEAP$
  $h.set\_once\_proc\_not\_fresh(p, f, st)$ **require**
    $f \in PROCEDURE \wedge f.is\_once$
  axioms
    $\neg h.set\_once\_proc\_not\_fresh(p, f, st).fresh(p, f.id)$
    $h.set\_once\_proc\_not\_fresh(p, f, st).stable(p, f.id) = st$
    $h.set\_once\_proc\_not\_fresh(p, f, st).stabilizer(p, f.id) = p$

Set not fresh

$$\frac{\begin{array}{l} f \in FUNCTION \wedge f.is\_once \\ \sigma.envs(p).top.names.has(result.name) \\ \sigma' \overset{def}{=} \sigma.set\_once\_func\_not\_fresh(p, f, false, \sigma.val(p, result.name)) \end{array}}{\Gamma \vdash \langle p :: \texttt{set\_not\_fresh}(f); s_p, \sigma\rangle \to \langle p :: s_p, \sigma\rangle}$$

Figure 5.2: Once routine stability clarification: status flags

$RETURN \triangleq \texttt{return}\ TUPLE\langle CHANNEL,\ FEATURE,\ PROC,\ BOOLEAN\rangle \mid$
$\texttt{return}\ TUPLE\langle CHANNEL,\ FEATURE,\ REF,\ PROC,\ BOOLEAN\rangle$;

Return (command)

$$\frac{\begin{array}{l}\sigma' \overset{def}{=} \begin{cases} \sigma.set\_once\_proc\_not\_fresh(p, f, true) & \text{if } f.is\_once \wedge snf \\ \sigma & \text{otherwise}\end{cases} \\ \sigma'' \overset{def}{=} \sigma'.pop\_env(p).revoke\_locks(q, p)\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{return}(a, f, q, snf); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ \sigma.passed(q)\ \texttt{then}\ \texttt{notify}(a)\ \texttt{else}\ \texttt{nop}\ \texttt{end}; s_p, \sigma''\rangle\end{array}}$$

Return (query)

$$\frac{\begin{array}{l}(\sigma', r_r') \overset{def}{=} \begin{cases} \text{if } r_r \neq void \wedge \sigma.ref\_obj(r_r).class\_type.is\_exp \wedge \sigma.handler(r_r) \neq q \\ \quad (\sigma^*, \sigma^*.last\_imported\_obj) \\ \quad\quad \textbf{where} \\ \quad\quad\quad \sigma^* \overset{def}{=} \sigma.import(q, r_r) \\ \text{otherwise} \\ \quad (\sigma, r_r)\end{cases} \\ \sigma'' \overset{def}{=} \begin{cases} \sigma'.set\_once\_func\_not\_fresh(p, f, true, r_r) & \text{if } f.is\_once \wedge snf \\ \sigma' & \text{otherwise}\end{cases} \\ \sigma''' \overset{def}{=} \sigma''.pop\_env(p).revoke\_locks(q, p)\end{array}}{\Gamma \vdash \langle p :: \texttt{return}(a, f, r_r, q, snf); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a, r_r'); s_p, \sigma'''\rangle}$$

Figure 5.3: Once routine stability clarification: return

The new status sequence prevents reading unstable values, but it can be problematic when used improperly. Consider a stabilizer p executing a fresh separate once function whose body has another call to f with a target on a different handler q. Once p reaches the call, it waits for q to return; however, q cannot execute f because it is not stable. To prevent this issue, the stabilizer must only call f on non-separate targets. If p calls a non-once routine h on q instead, q can callback f on p and thus get an unstable result. Processor p must therefore ensure that the once result is meaningful before passing any locks.

Clarification 5.2 (Once routines). Just after its first execution in a context, a once routine f is not fresh and *not stable*. After the execution of f, f becomes not fresh and *stable*. The processor that executes f for the first time is the *stabilizer*. Only the stabilizer can execute f while it is not stable. If f is a separate once function, the stabilizer must only call f on non-separate targets. Furthermore, it must ensure that the once result is meaningful before any calls with lock passing.
□

5.4 Lock passing for queries

A client only passes its locks when it performs a separate callback or when it has a lock that the supplier requires directly. Consequently, it does not pass its locks in a regular query call without arguments.

5.4.1 Flaw

This specification is problematic when the supplier requires one of the client's locks in a later call, as can be observed in the following code:

```
class A create make feature
  b: separate B
  c: separate C

  make
    do
      create c.make; create b.make (c)
      f (b, c)
    end

  f (b: separate B; c: separate C)
    local
      t: separate D
    do
      t := b.g
      c.k
    end
end

class B create make feature
  w: separate C

  make (c: separate C)
    do
      w := c
    end

  g: D
    do
      h (w)
      create Result.make
    end
  h (c: separate C)
    do
      c.k
    end
end

class C create make feature
  make
    do end

  k
    do end
end

class D create make feature
  make
    do end
end
```

Using Maude's **rew** command, a new root processor p_1 creates two objects on two new processors p_3 and p_5. It then starts executing the feature *f* and locks the two request queues:

p_1 :: orq: $(\{\}, \{p_3, p_5\})$ rrq: $(\{\}, \{\})$ rcs: $(\{\}, \{\})$ locked
p_3 :: orq: () rrq: () rcs: () locked
p_5 :: orq: () rrq: () rcs: () locked

In *f*, processor p_1 performs a query call to processor p_3. This query call is synchronous due to wait by necessity; however, it does not involve lock passing as processor p_3 does not require any locks from processor p_1 to execute *g*:

p_1 :: orq: $(\{\}, \{p_3, p_5\})$ rrq: $(\{\}, \{\})$ rcs: $(\{\}, \{\})$ locked
p_3 :: orq: $(\{\})$ rrq: $(\{\})$ rcs: $(\{\})$ locked
p_5 :: orq: () rrq: () rcs: () locked

Processor p_3 then tries to obtain processor p_5's request queue lock as it executes *h* (*w*). However, since processor p_1 still holds on to this lock while waiting for its query call to return, the system deadlocks. Maude stops because it cannot satisfy the premise of the lock operation.

5.4.2 Clarification

This design flaw can be addressed by always passing locks during a query call, as shown in Figure 5.4. This is not harmful because the client must wait anyway. The small change also facilitates separate callbacks where the feature call chain involves query calls (see Section 5.2).

$$
\frac{
\begin{array}{l}
\textsc{Call feature (query)}\\
q \overset{def}{=} \sigma.handler(r_0)\\
\bar{l} \overset{def}{=} \begin{cases} (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) & \text{if } q \neq p \\ (\{\}, \{\}) & \text{otherwise} \end{cases}\\
\sigma'_0 \overset{def}{=} \sigma\\
\forall i \in \{1, \dots, n\} \colon (\sigma'_i, r'_i) \overset{def}{=}\\
\quad \begin{cases}
\text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) \neq q\\
\quad (\sigma^*, \sigma^*.last\_imported\_obj)\\
\quad\quad \textbf{where}\\
\quad\quad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.import(q, r_i)\\
\text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) = q\\
\quad (\sigma^*, \sigma^*.last\_added\_obj)\\
\quad\quad \textbf{where}\\
\quad\quad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.add\_obj(q, \sigma'_{i-1}.ref\_obj(r_i).copy)\\
\text{otherwise}\\
\quad (\sigma'_{i-1}, r_i)
\end{cases}
\end{array}
}{
\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{call}(a, r_0, f, (r_1, \dots, r_n)); s_p, \sigma\rangle \rightarrow\\
\quad \langle p :: \texttt{issue}(q, \texttt{apply}(a, r_0, f, (r'_1, \dots, r'_n), p, \bar{l})); \texttt{wait}(a); s_p, \sigma'_n\rangle
\end{array}
}
$$

Figure 5.4: Query lock passing clarification

Clarification 5.3 (Lock passing). During a query call, a processor passes all its locks. □

5.5 Lock passing and asynchrony

While returning from a command call (see Figure 3.50 and Figure 5.3), a supplier p checks whether its client q has passed any locks , in which case it returns them and sends a notification.

5.5.1 Flaw

To check whether q has passed any locks, p queries $\sigma.passed(q)$. In case q has not passed any locks, the command call is asynchronous, and q can engage in further feature calls that do involve lock passing. If p returns while q has passed its locks to another processor, p synchronizes unnecessarily. The following code shows this scenario:

```
class A create make feature
  make
    local
      b: separate B
      c: separate C
    do
      create b.make; create c.make
      f (b, c)
    end

  f (b: separate B; c: separate C)
    do
      b.g -- Asynchronous
      c.h (b) -- Synchronous with lock passing
    end
end

class B create make feature
  make
    do end

  g
    local
      b: separate B
    do
      create b.make
    end
end

class C create make feature
  make do end

  h (b: separate B)
    do end
end
```

It is difficult to experience this flaw using the Maude built-in strategies as the strategy must ensure that the supplier returns while the client has passed its locks in another feature call. The following (truncated) custom strategy ensures the execution order with a client p_1, a supplier p_3 for the call $b.g$, and another supplier p_5 for the call $c.h\ (b)$:

```
... ;
*** Call b.g asynchronously.
parallelism{
  (match (Γ:Program |– proc(1) :: sl:List{Statement}, ic:Nat, σ:State)) ;
  callFeatureCommand} ;
... ;
*** Call c.h (b) synchronously with lock passing.
parallelism{
  (match (Γ:Program |– proc(1) :: sl:List{Statement}, ic:Nat, σ:State)) ;
  callFeatureCommand} ;
... ;
```

The strategy first lets p_1 perform the two feature calls. The first one is asynchronous, and the second one synchronous with lock passing. Hence, p_1 waits on channel a_5 for p_5. Since p_5 has not started the feature application yet, p_1 has not passed its locks yet:

```
⟨
    p₁ :: wait(a₅); . . . |
    p₃ :: apply(a₃, r_b, g, (), p₁, ({}, {})) |
    p₅ :: apply(a₅, r_c, h, (r_b), p₁, ({p₃, p₅}, {p₁}))
,
    locks:
        p₁ :: orq: ({}, {p₃, p₅}) rrq: ({}, {}) rcs: ({}, {}) locked
        p₃ :: orq: () rrq: () rcs: () locked
        p₅ :: orq: () rrq: () rcs: () locked
⟩
```

The strategy then lets both suppliers start their feature applications. For now, it only allows p_3 to continue all the way and return:

```
*** Start application of c.h (b).
parallelism{
  (match (Γ:Program |– proc(5) :: sl:List{Statement}, ic:Nat, σ:State)) ;
  applyFeatureNonOnceOrFresh} ;
```

```
... ;
*** Start application of b.g.
parallelism{
  (match (Γ:Program |– proc(3) :: sl:List{Statement}, ic:Nat, σ:State)) ;
  applyFeatureNonOnceOrFresh} ;
... ;
*** Return from application of b.g.
parallelism{
  (match (Γ:Program |– proc(3) :: sl:List{Statement}, ic:Nat, σ:State)) ;
  returnCommand} ;
... ;
```

Processor p_3 notices that p_1 has passed its locks (although to p_5 and not to p_3). Hence, p_3 tries to synchronize with p_1 over a_3. However, p_1 is not waiting for p_3 since it has not passed any locks to p_3:

$$
\begin{array}{l}
\langle \\
\quad p_1 :: \texttt{wait}(a_5); \ldots; \mid \\
\quad p_3 :: \texttt{notify}(a_3) \mid \\
\quad p_5 :: \texttt{execute\_body}(h, \{\}); \ldots \mid \\
, \\
\quad \text{locks:} \\
\qquad p_1 :: \ \text{orq}: (\{\}, \{p_3, p_5\}) \ \text{rrq}: (\{\}, \{\}) \ \text{rcs}: (\{\}, \{\}) \ \text{locked} \ \text{passed} \\
\qquad p_3 :: \ \text{orq}: () \ \text{rrq}: () \ \text{rcs}: () \ \text{locked} \\
\qquad p_5 :: \ \text{orq}: (\{\}) \ \text{rrq}: (\{p_3, p_5\}) \ \text{rcs}: (\{p_1\}) \ \text{locked} \\
\rangle
\end{array}
$$

The strategy then lets p_5 return as well:

```
... ;
*** Return from application of c.g (b).
parallelism{
  (match (Γ:Program |– proc(5) :: sl:List{Statement}, ic:Nat, σ:State)) ;
  returnCommand} ;
... ;
```

Processor p_5 also tries to synchronize with p_1 to notify p_1 of the returned locks. Since p_1 is actually waiting for p_5, the synchronization will succeed. Processor p_3 remains stuck trying to synchronize with p_1.

$$\begin{array}{l}
\langle \\
\quad p_1 :: \texttt{wait}(a_5); \ldots; \mid \\
\quad p_3 :: \texttt{notify}(a_3) \mid \\
\quad p_5 :: \texttt{notify}(a_5) \mid \\
, \\
\quad \text{locks}: \\
\qquad p_1 :: \ \text{orq}: (\{\}, \{p_3, p_5\}) \ \text{rrq}: (\{\}, \{\}) \ \text{rcs}: (\{\}, \{\}) \ \text{locked} \\
\qquad p_3 :: \ \text{orq}: () \ \text{rrq}: () \ \text{rcs}: () \ \text{locked} \\
\qquad p_5 :: \ \text{orq}: () \ \text{rrq}: () \ \text{rcs}: () \ \text{locked} \\
\rangle
\end{array}$$

5.5.2 Clarification

This flaw can be fixed by replacing the condition $\sigma.passed(q)$ in the `return` operation. The supplier p must check whether the client q has passed any locks *to* p. For this reason, the `return` operation must take the received locks $(\bar{l}_r, \bar{l}_c)$ and only synchronize if $(\bar{l}_r, \bar{l}_c)$ is not empty. Figure 5.5 shows the changes to the `return` variant for commands; the variant for queries must be adapted accordingly. The transition rules for feature applications (see Figure 3.46, Figure 3.51, and Figure 3.52) must be adapted to invoke `return` with the received locks.

$$\begin{array}{l}
RETURN \triangleq \\
\quad \texttt{return}\ TUPLE\langle CHANNEL,\ FEATURE,\ PROC, \\
\qquad TUPLE\langle SET\langle PROC\rangle,\ SET\langle PROC\rangle\rangle,\ BOOLEAN\rangle \ \mid \\
\quad \texttt{return}\ TUPLE\langle CHANNEL,\ FEATURE,\ REF,\ PROC, \\
\qquad TUPLE\langle SET\langle PROC\rangle,\ SET\langle PROC\rangle\rangle,\ BOOLEAN\rangle \ ;
\end{array}$$

Return (command)

$$\frac{\begin{array}{l}
\sigma' \overset{def}{=} \begin{cases} \sigma.set\_once\_proc\_not\_fresh(p, f, true) & \text{if } f.is\_once \wedge snf \\ \sigma & \text{otherwise} \end{cases} \\
\sigma'' \overset{def}{=} \sigma'.pop\_env(p).revoke\_locks(q, p)
\end{array}}{\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{return}(a, f, q, (\bar{l}_r, \bar{l}_c), snf); s_p, \sigma\rangle \rightarrow \\
\quad \langle p :: \texttt{provided}\ (\bar{l}_r, \bar{l}_c) \neq (\{\}, \{\})\ \texttt{then notify}(a)\ \texttt{else nop end}; s_p, \sigma''\rangle
\end{array}}$$

Figure 5.5: Lock revocation notification clarification

Clarification 5.4 (Lock passing). A supplier p of a command call must only synchronize with the client q, if q has passed any locks to p. □

5.6 Related work

Verification of programming models requires either testing or developing proofs. This section discusses related work from both angles.

5.6.1 Testing of programming models

Ostroff et al. [173] present an operational semantics for a SCOOP subset and use it in a model checker to verify small SCOOP programs. In contrast, our approach focuses on discovering flaws in the SCOOP model and not on program verification. Although their formal semantics covers some of the most significant aspects of SCOOP, it falls short of describing a number of other critical aspects (see Table 5.2): selective locking, lock passing, separate callbacks, once routines, expanded objects with the import operation, explicit processor specifications, as well as postcondition and invariant evaluations. Further, their formal semantics introduces an unnecessary limitation for queries: a processor *p* executing a query must not perform any calls to other processors.

Brooke et al. [37] present an operational semantics of SCOOP in CSP [91] and use a model checker to test the model with example programs. They explore different lock passing mechanisms and different strategies to release locks. They conclude that a supplier can unlock as soon as its client stops using it. Compared to our formal semantics, which comprehensively models SCOOP, the CSP model focuses only on the core SCOOP concepts (see Table 5.2) – automatic locking and unlocking, preconditions with wait semantics, wait by necessity, and lock passing – and does not cover other important aspects of SCOOP – selective locking, separate callbacks, once routines, expanded objects with the import operation, explicit processor specifications, as well as postcondition and invariant evaluations. Since the CSP model does not support expression evaluations, it simulates precondition evaluations with two events per feature call, one for a satisfied precondition and one for a failed precondition.

Ellison [65] uses K [187] to describe an operational semantics of C [111, 113] with support for threads. He gains confidence in his semantics by testing it against a GCC test suite. For SCOOP, we had to compose our own test suite because no systematic test suite was available. Ellison uses Maude's LTL model checker and search command to show mutual exclusion in Dekker's algorithm and deadlock-freedom in a dining philosophers implementation. His approach is similar to our approach, with the major difference that he does not use it to develop and clarify a concurrency model.

AlTurki and Meseguer [7,8] implement a structural operational semantics [179] and an equivalent but more efficient reduction semantics [71] for Orc [157, 169] using Maude's real time extension [171]. Orc is a concurrency model and pro-

Table 5.2: Comparison of formal semantics for SCOOP. The marks indicate the supported aspects, where ● stands for complete support, and ◐ stands for partial support.

semantics	locking, preconditions, and unlocking	wait by necessity	lock passing	separate callbacks	once routines	import	explicit processor specifications	postconditions and invariants	failures	passive processors
Morandi (this thesis)	●	●	●	●	●	●	●	●	●	●
Ostroff et al. [173]	◐	●								
Brooke et al. [37]	◐	●	●							
Bailly [17]	◐	●								

gramming language for task orchestration. It provides constructs to orchestrate the concurrent execution of services while managing timeouts, priorities, and failures. AlTurki and Meseguer use Maude's search command and the LTL model checker to verify safety and liveness properties of Orc programs such as a dining philosopher implementation and an online auction system. They extend [9] their structural operational semantics to also model distribution using Maude's support for TCP sockets. Using the distributed semantics, they apply time-bounded LTL model checking to a distributed version of the online auction system. However, they do not use their semantics to clarify or develop Orc. Since SCOOP does not have any real-time constructs, we did not use Maude's real time extension.

Meredith et al. [143] use Maude to develop an operational semantics for a subset of a Verilog [101] precursor. They use the semantics to find a bug in a Verilog implementation by comparing the two outputs of a Verilog program.

Sperber et al. [199] define an operational semantics for a Scheme [100] subset in PLT Redex [70]. Their semantics comes with an extensive test suite. Although they do not point out any clarifications, the test suite is likely to have been helpful in the development of the semantics.

Palmer et al. [176] use TLA+ [129] to introduce a formal semantics for a subset of an MPI [146] precursor. They show a number of issues in the informal description, discovered during formalization, and establish independence theorems for MPI operations. Furthermore, they present a tool that translates an MPI program with embedded assertions into TLA+, invokes the TLC model checker [223], and reports the results in terms of the program code. They use their tool to verify a few MPI programs. Li et al. [131] provide a more comprehensive MPI semantics in TLA+ and discuss issues in the MPI description, found during formalization. They use the semantics to prove the correctness of the completes-before relation [207]. By model checking a comprehensive test suite with TLC, they demonstrate the semantics' self-consistency and adherence to the informal description, but they do not report any flaws in MPI found during testing.

Mousavi et al. [159, 160] develop an operational semantics in Maude for a subset of Reo [12], a coordination model with basic and composed connectors that link nodes to each other. They present a number of connectors and use Maude's LTL mode checker to verify one of them, but they do not report any findings on the coordination model.

Verdejo and Martí-Oliet [209] describe various formal semantics of programming models implemented in Maude. Some of those semantics have intricate features, which make the implementation challenging, but they are still very compact compared to SCOOP. Thatia, Sen, and Martí-Oliet develop a formal semantics [203] of an asynchronous version of the π-calculus [156].

Farzan et al. [68, 69] use Maude to introduce an operational semantics for a source code and bytecode subset of a Java [82, 132] precursor. To alleviate analysis, a front-end called JavaFAN translates programs and properties into Maude. It then invokes Maude's execution capabilities, i.e., the rewrite or frewrite commands to find one possible execution and the search command or the LTL model checker to prove or disprove a property. Finally, JavaFAN presents the results in a readable format. In contrast to our work, Farzan et al. mostly check properties of Java programs and do not focus on the development of a concurrency model.

Attali et al. [14] present an operational semantics of Eiffel// [47] in CENTAUR [31]. A natural semantics [120] specifies inheritance and dynamic binding, and a structural operational semantics [179] specifies the remaining aspects. An interactive environment serves as the front-end for the semantics. The front-end supports developing programs, changing the semantics, and browsing through different executions. It visualizes the executions either graphically or textually. While the front-end has a built-in deadlock detection, the authors do not report any testing results.

Kulas and Beierle [126] develop an operational semantics for Prolog [106, 108, 110, 114] in Maude. However, they do not use any test programs to clarify the programming model.

Testing has also been applied to memory models. Batty et al. [20] present an axiomatic semantics for a draft version of the C++ memory model [112] in Isabelle/HOL [165] along with definitions for data races, unsequenced races, and indeterminate reads. They develop a tool called CPPMEM that finds all potential executions of a program, filters the ones allowed by the axiomatic semantics, and reports any flaws. During the formalization effort and using CPPMEM, they report a number of flaws in the informal description of the memory model. To validate their semantics of the memory model, they prove semantic preservation of a compilation scheme from a program with their semantics to a program with the x86-TSO semantics [174, 194].

Since CPPMEM searches exhaustively for potential executions, its usage has been limited to small programs. To skip the exhaustive search, Blanchette et al. [23] propose an improved version of Nitpick, Isabelle's SAT-based model finder, and an adapted axiomatic semantics instead of CPPMEM. They show that this approach scales better to larger programs.

Torlak et al. [204] also use a SAT solver to develop MemSAT, another tool to explore memory models. MemSAT takes an axiomatic semantics of a memory model and a test program with assertions. It returns executions in which both the assertions and the memory model axioms are satisfied. The authors used MemSAT to check existing memory models against test cases and report on subtle discrepancies. Tools like MemSAT, CPPMEM, and earlier ones [221, 222] do not comprehensively analyze a concurrency model as they focus on the memory model; hence, they are not suited to study full-fledged concurrent programs.

5.6.2 *Proofs for programming models*

The maturity of reasoning frameworks has led to a growing interest in proofs for large formal semantics of programming models and the compilation process. Klein and Nipkow [122] use Isabelle/HOL [165] to present an operational semantics for a single-threaded Java subset with its bytecode and prove type safety. They also formalize the source to bytecode compiler and prove that the compilation preserves the semantics and well-typedness. Lochbihler [136] extends this work with an axiomatic semantics of the Java memory model and all of Java's concurrency features in a precursor of the informal description [82, 132] except deprecated methods, timing-related methods, vendor-specific extensions, and (discouraged) spurious wake-ups. He proves type safety up to deadlocks, sequential consistency for correctly synchronized programs, and the existence of a legal execution for every well-formed program. For the compilation process, he shows semantic preservation including deadlocks and data races. To demonstrate that the operational semantics faithfully follows the informal description, he uses Isabelle's code generator to extract a source code and byte code interpreter as well as a compiler. He uses the interpreter to run a test suite and compare the results to the ones from the Java reference implementation. Using this technique, he finds a flaw with binary operators, confirming that testing of executable formal semantics is beneficial even after proving important properties. In contrast to Lochbihler's approach, our approach focuses on prototyping. As testing gives a faster feedback than fully formal proofs, our approach emphasizes testing to improve an executable formal semantics until it meets the developers expectations.

Boldo et al. [25] use the Coq [22] proof assistant to prove that the CompCert [130] compiler preserves the floating-point semantics of a C [111, 113] precursor. Similarly, Batty et al. [19] use the theorem prover HOL4 [197] to prove the correctness of the proposed compilation schemes for the C++ [112] load and store concurrency primitives.

Wasserrab [211] uses Isabelle/HOL [165] to implement an operational semantics for a large subset of a C++ [112] precursor. He proves type safety in presence of multiple inheritance. Just like Lochbihler [136] he uses Isabelle's code generator to obtain an interpreter. However, he does not report on any testing results. Similarly, Norrish [166, 167] uses HOL to provide operational semantics for a large subset of a C [111, 113] precursor and a large subset of a C++ [112] precursor. For C, he proves type safety and progress; for C++, he proves that a test suite behaves as expected.

Stärk et al. [200] introduce an operational semantics for a large source code and bytecode subset of a Java [82, 132] precursor. The semantics is based on ASMs [29, 85] implemented in AsmGofer [191]. Stärk et al. also formalize the source to bytecode compiler and the bytecode verifier, but they only formalize a simplified memory model. A number of manual proofs show type safety, semantic preservation during compilation, soundness and completeness of the bytecode verifier, as well as soundness of the thread synchronization mechanism. Further manual proofs show under what assumptions a compiled program can be successfully verified and executed. Also based on ASMs, Fruja et al. [28, 75, 119] use AsmL [86] to present a formal semantics for a subset of a C# [104] precursor. During the formalization effort, they found a number of weak points in the informal description. However, they neither prove any theorems nor do they report on any testing results.

Numerous non-executable formal semantics facilitate proofs for programs and programming models. Wehrmann et al. [213] present a timed operational semantics and an event-based denotational semantics for Orc [157, 169]. They show the equivalence of the two semantics as well as many other properties for expressions [212].

Cenciarelli et al. [50] define an operational semantics for a subset of a Java [82] precursor. They combine the operational semantics with a denotational one by extending execution configurations with a history of events that took place. An axiomatic semantics uses the events to restrict the result configurations reachable from a start configuration. They show that the formal semantics only allows executions that are also allowed by the informal description.

Ábrahám [1] also presents an operational semantics for a subset of a Java [82] precursor, focusing on dynamic thread creation and re-entrant monitors. The operational semantics is the basis of a sound and complete proof system for invariants. For a given invariant, a tool generates a set of verification conditions, to be proven in PVS [175].

Bailly [17] presents an operational semantics with a sound and complete proof system for a subset of SCOOP. He uses the proof system to establish safety properties of SCOOP programs. The operational semantics does not serve as a formal reference of SCOOP because it does not cover selective locking, lock passing, separate callbacks, once routines, expanded objects with the import operation, explicit processor specifications, as well as postcondition and invariant evaluations (see Table 5.2). Furthermore, it has limited support for expression evaluations, and it assumes that clients lock their suppliers in conditional critical regions rather than at the beginning of a feature application.

5.7 Discussion

The testing effort revealed four flaws in SCOOP. The two flaws in Section 5.2 and Section 5.4 can be found with the Maude built-in execution strategies because they do not depend on a particular schedule. The other two flaws in Section 5.3 and Section 5.5 only appear under particular schedules, not reproducible with the deterministic built-in strategies. Thus, these flaws can only be found by providing a custom strategy or by applying an exhaustive strategy. The latter strategy is in general infeasible due to state explosion. The exhaustive search can, however, be successful if it finds the right schedule early.

6 Asynchronous exceptions

The previous chapters showed the benefits of applying the prototyping method to an existing model. The benefits do not end there: the method also facilitates the development of model extensions. This and the next chapter apply the method to two new mechanisms. This chapter extends SCOOP with a mechanism for asynchronous exceptions.

An exception is the result of a *failure* – the inability of the supplier to fulfill its *promise* towards its client. The supplier *raises* the exception, and some processor must *propagate the exception*, i.e., react to it by changing the control flow, and subsequently *handle* it by executing a dedicated sequence of instructions. The *failure context* consists of two parts: the *failed context* is the supplier's context, and the *responsible context* is the context in which the client made the invocation. An *asynchronous exception* is an exception raised during an asynchronous feature call, e.g., a feature call to a command without lock passing. Due to the asynchrony, the client might already have continued its execution and no longer be in a position to propagate the exception. An asynchronous exception mechanism specifies where, when, and how the asynchronous exception propagates.

For SCOOP, two mechanisms [13,35] for asynchronous exceptions have been proposed. However, these mechanisms are not fully satisfactory (see Section 6.7). To develop a new mechanism, this chapter first classifies and discusses existing approaches for asynchronous exceptions. It then presents the *accountability* framework to describe, comprehend, and compare asynchronous exception mechanisms: only within a well-defined range is the supplier obliged to report failures, and it only does so if it is *held accountable*. Based on the classification and the accountability framework, it then derives a new mechanism for SCOOP.

6.1 Classification of approaches

The literature contains a wide variety of asynchronous exception mechanisms. Some mechanisms support multiple approaches to combine the advantages and compensate for the disadvantages of these approaches. Figure 6.1 shows a classification of approaches. On the top level, the classification differentiates between

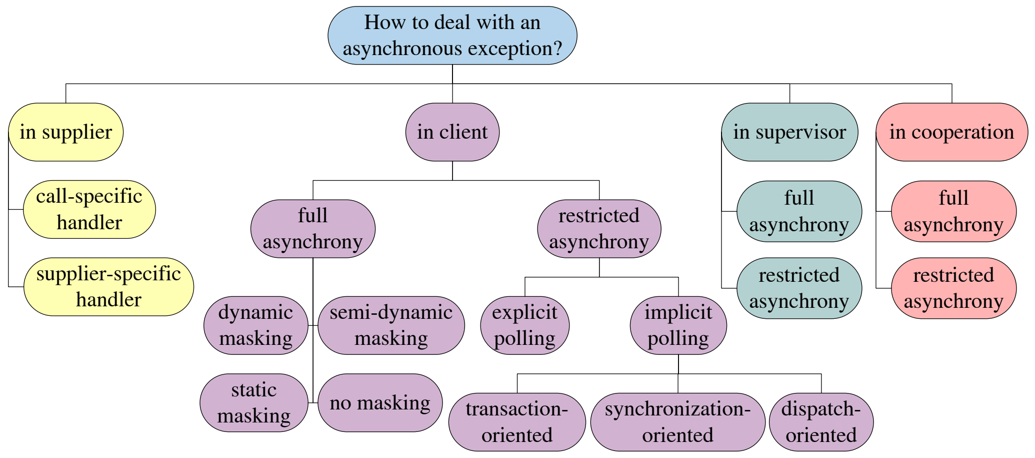
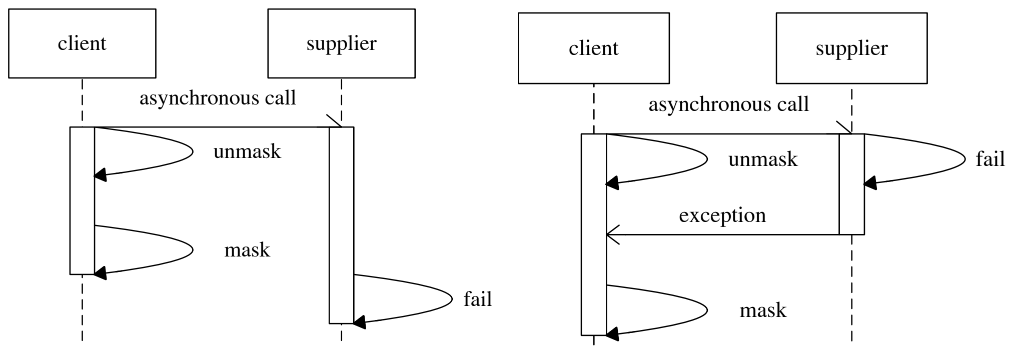


Figure 6.1: Classification of approaches to deal with an asynchronous exception. The figure omits repeating the subdivisions for the supervisor-based and cooperation-based approaches.

locations where an asynchronous exception can propagate. In case *the supplier propagates* the exception, the client does not have to expect an exception; it can be assured that the supplier will take care of any exceptions. Moreover, the supplier can handle the exception in the failed context. As a disadvantage, the client has no direct way of telling whether its call succeeded. Furthermore, the supplier might not always be able to draw the right conclusions on behalf of the client, as it does not have access to the responsible context. The handler can either be defined in the supplier or it can come from the client via an argument. Supplier-specific handlers are common: any mechanism where a supplier can deal with an exception in a local handler falls into this category; one example is SaGE [58]. Call-specific handlers are less common, but there are mechanisms, such as JR [170], using this approach. Call-specific handlers contain clean-up code specified by the client; in general, however, the client does not know how to clean up the supplier without violating information hiding principles. Supplier-specific handlers do not have this problem.

The case that the *client propagates* the exception supports a response in the responsible context, but in turn the client does not have access to the failed context.

Table 6.1 lists the mechanisms that support this kind of propagation. The classification differentiates two cases: full asynchrony and restricted asynchrony. With *full asynchrony*, the exception can propagate anytime unless the client masked the exception. This approach is well-suited for reactive systems. However, programmers have to program under the assumption of interruptibility. Furthermore, there is a risk of race conditions: the timing of the propagation depends on the relative speed between the client and the supplier (see Figure 6.2).



(a) The client does not propagate the exception. (b) The client propagates the exception.

Figure 6.2: Race condition with full asynchrony. Figure 6.2a and Figure 6.2b show two executions for two different relative speeds between the client and the supplier. In Figure 6.2a, the client does not propagate the supplier's exception because the client has masked exceptions. In Figure 6.2b, the client propagates the exception because it has not yet masked exceptions.

For masking, Krischer [125] differentiates several possibilities:

- *Dynamic masking*. Propagation is enabled until it is disabled by an opposing call. One example is Haskell [140] with its block and unblock operations.
- *Semi-dynamic masking*. Propagation is enabled only within a dedicated program block. Examples include Haskell [140] with its scopes, μC++ [40], and SR [96].
- *Static masking*. Propagation is enabled within a dedicated program block, but only when the client is statically in the block. Examples include the mechanism proposed by Krischer [125] and RTSJ [26].
- *No masking*. Propagation is always enabled. Examples include SaGE [58] with its client-based handlers, Erlang [66] with its exit signals over process links, Arche [116] where a client propagates exceptions as a result of incoming method calls, and ARMI [183] with its callback mechanism.

Table 6.1: Classification of mechanisms with propagation in the client. The marks indicate the supported approaches. The horizontal lines group mechanisms with a similar focus.

mechanism	full asynchrony				restricted asynchrony			
	dynamic	semi-dynamic	static	no	explicit polling	implicit polling		
	masking	masking	masking	masking		transaction	synchronization	dispatch
Haskell [140]	X	X						
μC++ [40]		X			X		X	
SR [96]		X						
Krischer [125]			X		X			
RTSJ [26]			X					
SaGE [58]				X				
Erlang [66]				X	X			
Arche [116]				X				
ABCL/1 [98]					X		X	
Modula-3 [163]					X		X	
Argus [133]						X	X	
Avalon/C++ [216]						X		
busy processor [13]						X		
X10 [188]							X	
RMIX [127]							X	
Fortress [3]							X	
Cilk Plus [102]							X	
ProActive [48]					X		X	
Rintala [186]							X	
ARMI [183]				X			X	
failed object [35]							X	X

With *restricted asynchrony*, the exception can only propagate when the client polls the supplier, thus alleviating the interruptibility issue. This comes at the price of a delay between the moment the supplier raises the exception until the client propagates it, which might be unacceptable for reactive systems. Moreover, polling adds to the execution time even when there are no exceptions. This polling overhead can be diminished, in case the client waits for the supplier anyway to first finish its workload upon polling; as a further advantage of this, race conditions cannot occur because the supplier always knows about its failures before answering. The classification differentiates between *explicit polling*, where the client polls using dedicated polling constructs, and *implicit polling*, where the client polls in the course of executing constructs that are not solely dedicated to exceptions. Mechanisms that offer explicit polling include μC++ [40], Krischer's mechanism [125], Erlang [66] with its exit signal traps, ABCL/1 [98] where the client can wait for exception messages, Modula-3 [163] with its alert tests, and ProActive [48] with its exception queries and try block barriers. With the explicit polling approach, it is clear when the client polls. By restricting the propagation to dedicated points, programmers can first focus on the normal behavior and deal with exceptions in a second step. Furthermore, programmers are free to synchronize as often as necessary, thus mitigating unwanted delays. As a drawback, the language or its library must introduce a primitive for this purpose. The implicit polling approach does not suffer from this; programmers must however be aware of the constructs that imply polling. To poll implicitly, there are three categories:

- *Transaction-oriented polling*. The client polls at specific points in a transaction. Examples include Argus [133] with its coenter statement and Avalon/C++ [216] with its costart statement. Arslan and Meyer's busy processor approach [13] for SCOOP also falls into this category: the client polls for exceptions while locking the supplier. This approach fits well for languages with transactions; furthermore, it is clear when the client must expect an exception. However, transaction-oriented polling alone is too coarse-grained for scenarios where the client has to know about exceptions at a location that is not related to a transaction.

- *Synchronization-oriented polling*. The client polls upon synchronization with the supplier. Examples include Modula-3 [163] with its alert joins and alert wait, Argus [134] with its synch statement, X10 [188] with its finish statement, RMIX [127], Fortress [3] with its val method, and Cilk Plus [102] with its sync statement. One example for SCOOP is Brooke and Paige's failed object mechanism [35]: after an object caused an exception, it is marked as failed; any future client propagates an exception upon a synchronous call. Further examples include languages with futures:

μC++ [40], ABCL/1 [98], Argus [134] with its promises, ProActive [48], Rintala's C++ futures [186], and ARMI [183] with its delayed delivery mechanism. A *future* represents a result that a supplier is computing asynchronously. When the client needs the result, it accesses the future. This forces the client to wait until the result is available or the supplier raises an exception. In the latter case, the client propagates the exception. With the synchronization-oriented approach, the client polls when it has to wait anyway; hence, it can diminish parts of the polling time. The approach also helps to mitigate unwanted delays: programmers are free to poll as often as necessary. Race conditions cannot occur with this approach because the supplier finishes its workload upon polling, and hence, it always knows about its failures before answering. On the downside of synchronization-oriented polling, programmers have to remember all possible implicit synchronization points.

- *Dispatch-oriented polling*. The client polls whenever it dispatches a message to the supplier. Brooke and Paige's failed object mechanism [35] also falls into this category; the client propagates an exception upon an asynchronous call as well. This approach helps to mitigate unwanted delays because every dispatch is a poll point. However, this approach can introduce race conditions because a dispatch does not wait for the supplier to finish the issued workload. Furthermore, programmers have to be aware that each call causes polling.

Besides using the supplier or the client, a *supervisor* can be used to propagate the exception. The supervisor monitors the supplier to act upon exceptions. This approach supports the separation of normal behavior from exceptional behavior. Supervisors neither have access to the responsible context nor to the failed context; they can, however, have a global view on the system, as they can be supervising multiple suppliers. Similar to propagation in the client, the supervisor can be based on *full asynchrony* or *restricted asynchrony*. In μC++ [40], a supervisor comes into place when a supplier raises an exception in a task different from the client. Such a supervisor is in general not solely dedicated to monitoring; hence, μC++ makes use of full asynchrony with semi-dynamic masking and restricted asynchrony with explicit polling. In Erlang [66], supervisors are dedicated to monitoring; hence, they are based on full asynchrony with no masking. The same is true for Ajanta [206] with its guardian agents. In Arche [116], a client specifies supervisors upon calling; the supervisors treat the exceptions as incoming method calls. In ABCL/1 [98], each call specifies a number of supervisors through complaint destinations; the supervisors use explicit polling to propagate exceptions. Modula-3 [163] supports supervisors with explicit polling and implicit polling.

Finally, the exception can propagate in an entire *cooperation* with several threads performing a job together. CA actions [185,219] provide a general framework for coordinated exception handling. This approach supports a collaborative, yet member-specific response in the responsible context, but the coordination is costly and the members do not have access to the failed context. As with the propagation in clients and supervisors, the members can propagate the exception using *full asynchrony* or *restricted asynchrony*. For example, the Guardian model [154] follows the fully asynchronous approach.

6.2 Accountability framework

Each of the mechanisms discussed in Section 6.1 differs from the others in some way. On an abstract level, however, each of them addresses the same set of essential questions. This section introduces the accountability framework, identifying these questions and providing concepts to answer them. In governance, Schedler [189] defines *accountability*:

> A is accountable to B when A is obliged to inform B about A's (past or future) actions and decisions, to justify them, and to suffer punishment in the case of eventual misconduct.

In essence, accountability is the obligation to inform about a failure and to take the consequences. It not only applies in governance, but also to threads. Accountability describes a supplier's obligation to inform about its failure after a failed asynchronous invocation. The supplier is either accountable to its client, to a supervisor, to a cooperation, or to no one, as it is the case with propagation in the supplier. Definition 6.1 introduces accountability for threads. It defines concepts that appear in all mechanisms and uses them to identify the essential questions that each mechanism answers individually. The concepts and the questions form the accountability framework.

Definition 6.1 (Accountability framework). Consider a *client* that expects a *supplier* to perform some activities on its behalf. When the supplier fails, it performs a *local reaction*. An *observer* performs a *remote reaction* in response to the failure. The supplier is *accountable* to the observer for one of its activities if it is guaranteed to report any inability to fulfill the promise about the activity made. When the observer asks the supplier to report a failure, the observer is said to *hold* the supplier *accountable*. The guarantee is called an *accountability*. While the guarantee lasts, the accountability is said to be *alive*; afterwards it is *expired*. To instantiate this conceptual framework for a particular mechanism, the following questions must be answered:

- What is the supplier's local reaction?
- Who is the observer?
- When does an accountability become alive and when does it expire?
- When does the observer hold the supplier accountable?
- What is the observer's remote reaction?

□

The accountability framework and the classification from Section 6.1 can be used as a guide to develop new mechanisms: the accountability framework defines the primary questions to be answered; the classification discusses possible answers from the literature. The framework can also be used to describe existing approaches.

6.3 Accountability framework instantiations

To demonstrate the generality of the accountability framework, this section describes three diverse mechanisms. Section 6.3.1 instantiates the framework for μC++ [40], Section 6.3.2 for Erlang [66], and Section 6.3.3 for SaGE [58].

6.3.1 Accountability for μC++

μC++ [40] extends C++ [112] with language constructs for concurrency. A *task* can asynchronously spawn another task and retrieve results from it over a future. The supplier fails by executing a *throw* or *resume* statement.

What is the supplier's local reaction? Programmers can enclose code blocks with two kinds of handlers: *termination* handlers and *resumption* handlers. When a supplier executes a throw statement, it continues the execution in the nearest enclosing termination handler; after the supplier completes, it continues after the termination handler. When a supplier executes a resume statement, it executes the nearest enclosing resumption handler; after the supplier completes, it returns to the statement after the resume statement. A resume statement can optionally specify a different task to be informed about the failure, in which case the supplier remembers its failure and continues. In case the supplier issued a future to a client, it can additionally use the future to remember its failure.

Default handlers exist for the case no explicit termination or resumption handler has been defined. The default termination handler terminates the program;

the default resumption handler initiates a search for a termination handler from the point of the initial resume statement.

Who is the observer? An observer comes into place when the supplier executes a resume statement that specifies another task; the specified task becomes the observer. This task can either be the client, or any other task that is dedicated to supervise the supplier. A task also becomes an observer when it receives a future from the supplier.

When does an accountability become alive and when does it expire? A supplier's accountability becomes alive with the creation of the supplier, and it expires when all its observers terminate. Consequently, the supplier might remain accountable after it terminates.

When does the observer hold the supplier accountable? An observer holds a supplier accountable when executing a block where masking is disabled, when executing a poll statement, or when querying a future from the supplier.

What is the observer's remote reaction? When the observer learns of a supplier's failure by accessing a future, it executes the nearest enclosing termination handler; the observer then continues after the termination handler. In all other cases, the observer behaves as if it executed a resume statement, i.e., it executes the nearest enclosing resumption handler and then returns to the statement after the resume statement.

6.3.2 Accountability for Erlang

Erlang [66] is a functional programming language whose concurrency support is based on the *actor model* [89]. An actor is a thread that responds to incoming asynchronous messages from other actors; in response to a message, the actor can either create other actors, send messages to other actors, or determine new behavior for future messages. In Erlang, actors are termed processes, and these processes can be linked to each other.

What is the supplier's local reaction? When a supplier fails, it remembers the failure, terminates, and waits until it is held accountable.

Who is the observer? Dedicated supervisors are responsible for monitoring processes. Each supervisor has a list of *child specifications* that specifies the processes to be monitored. A supervisor can also monitor another supervisor, permitting a supervision tree. In addition to supervisors, a linked client is also an observer.

When does an accountability become alive and when does it expire? An accountability becomes alive with the creation of a process; it expires with the termination of all observers.

When does the observer hold the supplier accountable? A supervisor holds its monitored processes permanently accountable; it uses full asynchrony without masking. A linked client can have a *trap* to convert the failure of a supplier into an *exit message*. In case a linked client has a trap, it holds the supplier accountable when it responds to the exit message; otherwise, it holds the supplier permanently accountable.

What is the observer's remote reaction? The list of child specifications determines for each monitored process how the supervisor will react upon a failure; possible reactions are restart or shutdown. For both restart and shutdown, there are a number of strategies to choose from. For instance, the supervisor can either restart just the failed process or all monitored processes. A linked client with a trap processes the supplier's failure as a message whereas a linked client without a trap fails.

6.3.3 Accountability for SaGE

SaGE [58] is an exception mechanism for languages using the *active object* pattern [59]. The active object pattern provides a solution to decouple an invocation in one object from the resulting execution in another object by giving an own thread of control to each object; this pattern is often used to implement the actor model [89]. In SaGE, each object scans its received messages. When an object decides to accept a request in a message, it spawns a *service* that processes the message concurrently. The service, i.e., the client, can then send messages to other objects, causing further services, i.e., the suppliers, to be created.

What is the supplier's local reaction? Programmers can attach handlers to services or objects. When a supplier fails, it first searches for a matching handler attached to itself. If no such handler exists, the supplier searches for a matching handler attached to its owner object. In case the supplier finds a handler in this way, it executes the handler. If the supplier does not fail during the execution of the handler, it terminates along with its own suppliers. If the supplier fails during the execution of the handler, or if it does not find a handler, it remembers the failure and waits until it is held accountable. After it is held accountable, the supplier terminates along with its suppliers.

Who is the observer? The observer is the client that called the faulty supplier.

When does an accountability become alive and when does it expire? An accountability becomes alive when an object creates a service. The accountability expires when the service terminates.

When does the observer hold the supplier accountable? A client holds a called supplier permanently accountable, i.e., even after the client terminates. The client uses full asynchrony without masking.

What is the observer's remote reaction? In addition to handlers attached to services and objects, programmers can also attach handlers to requests. When a client learns about the failure of a supplier, it first searches for a matching handler that is attached to the corresponding request. If no such handler exists, the client evaluates a *resolution function*. The purpose of the resolution function is to aggregate and filter exceptions from different suppliers. If an exception remains, the client treats the supplier's failure as its own.

6.4 Informal description

The proposed exception mechanism for SCOOP builds on an existing mechanism for non-concurrent programs. This section first presents the non-concurrent mechanism and then uses the accountability framework to develop the concurrent one for SCOOP.

6.4.1 Exceptions in non-concurrent SCOOP programs

The presented mechanism for SCOOP relies on the *rescue-retry* approach for non-concurrent programs [150]. While this approach originates in Eiffel, languages such as Ruby [99] or Jass [18] have also adopted it. In the rescue-retry approach there are two possible responses to a failure:

- *Organized panic*. The supplier admits that the promise cannot be fulfilled. It brings all affected objects to a consistent state and reports the failure by passing the exception to its client.
- *Resumption*. The supplier attempts to fix the reasons for the failure and retries the execution.

Programmers decide on the appropriate response by providing a rescue clause (**rescue** keyword) for each feature; features without an explicit rescue clause implicitly have an empty one. Whenever the supplier fails, it executes the rescue clause to put affected objects back into a consistent state. If the supplier reaches the end of the rescue clause, it reports the failure to the client (organized panic). The supplier can leave the rescue clause by executing a retry instruction (**retry** keyword), in which case it restarts the feature execution (resumption). Before retrying, the supplier must ensure that the new attempt is more successful, e.g., by trying a different strategy.

As an example, consider an application that explores a search space to find solutions to a problem. A controller triggers two concurrent searchers. A log records the solutions.

```
class CONTROLLER feature
  start (
    first_searcher: separate SEARCHER;
    second_searcher: separate SEARCHER;
    log: separate LOG
  )
      -- Use the searchers to search for a solution; log one of the solutions.
    do
      first_searcher.search; second_searcher.search -- Search concurrently.
      log.add_entry (first_searcher, second_searcher) -- Record one of the solutions.
    end
end
```

The class *SEARCHER* has a feature *search* to find a solution using a random search with a new seed. If the new seed is invalid, the searcher raises an exception, and the rescue clause restores the seed before reporting the failure. The postcondition expresses the promise.

```
class SEARCHER feature
  seed: INTEGER -- The seed used in the random search.
  solution: STRING -- The result.

  search
      -- Search for a solution.
    do
       -- Get the seed from atmospheric noise.
      seed := atmospheric_noise
      if seed >= 0 then
        solution := random_solution (seed) -- Search.
      else
        raise -- Fail.
      end
    ensure
      not equal (solution, Void)
    rescue
      seed := 0 -- Restore consistency.
    end

invariant seed >= 0 -- The consistency criterion.
end
```

6.4.2 Accountability for SCOOP

The main goal of SCOOP is to provide concurrency mechanisms that are simple to understand and to use [150, 164]; the asynchronous exception mechanism must be designed in the same spirit. This is part of the following requirements:

- *Comprehensibility*. The mechanism must be easy to understand.
- *Compatibility*. It must be compatible with the existing mechanism for non-concurrent programs (see Section 6.4.1).
- *Consistency*. It must guarantee the consistency of a failed supplier.

What is the supplier's local reaction? To satisfy the compatibility and the consistency requirements, the supplier must execute its rescue clause to reestablish its consistency, as explained in Section 6.4.1. The rescue clause is a supplier-specific handler. Without a retry instruction, the supplier first determines whether the feature call is asynchronous or synchronous. For an asynchronous call, the supplier remembers the failure and cleans up. The supplier purges any remaining feature requests from the client since they depend on the postcondition of the failed call. It also releases its obtained locks and returns any retrieved locks. It then waits until it is held accountable or until its accountability expires. For a synchronous call, the supplier notifies the client. It neither remembers the failure nor purges any feature requests; hence, it remains available for further work. This definition ensures compatibility.

Who is the observer? Propagation in a supervisor or in a cooperation would not be suitable for SCOOP because these approaches are not compatible with the existing exception mechanism for non-concurrent programs; by definition, a non-concurrent program cannot have a supervisor or cooperation members working in parallel to the main execution. The client is a better observer. Using the client as the observer, the mechanism meets the compatibility requirement because the non-concurrent mechanism also displays a division of labor between the supplier and the client.

When does an accountability become alive and when does it expire? To answer this question, it is helpful to consider when a client *can assume the promise* of a supplier fulfilling the postcondition of an issued feature call: this is the case as long as the supplier's request queue is locked; as soon as the request queue is unlocked, the client no longer has any guarantees because another processor could have modified the supplier. A client *relies on a promise* of a supplier by performing a synchronous feature call while it can still assume the promise. We argue that a supplier's failure only matters when the client relies on the promise of the supplier. This view is compatible with the view on asynchrony in SCOOP: the supplier is required to establish the promise of an asynchronous feature call not immediately, but instead when the client relies on the promise [150, 164]. Therefore, a failure does not matter anymore when the client can no longer assume the promise. For the proposed mechanism, this means that a supplier is accountable as long as its request queue is locked. By letting an accountability expire, SCOOP programs can become more fault-tolerant. Because a supplier always restores the consistency of a failed object, this object can safely be accessed again.

Lock passing complicates the situation. When a client p passes a lock on a supplier to a new client p', p *transfers* the supplier's accountability to p'; when p' returns the lock to p, p' also returns the accountability. For the duration of lock passing, p' is the observer. Accountability transfer extends the responsible context to include all feature executions along the lock passing chain. When a client p passes its call stack lock to another processor p', p' becomes an observer with a new accountability. This accountability expires when p' returns the call stack lock.

One question remains: does it make sense for a client not to rely on a promise? It does, for example, when the client spawns multiple suppliers but only wants the results from some of them. In Section 6.4.1, the log might not care about the second searcher if the first searcher found a satisfying solution. In absence of a cancellation mechanism, the log must ignore the second searcher by not synchronizing with it.

Mechanisms using futures, e.g., ProActive [48], also allow an accountability to expire: when the client does not query the future, the accountability expires. For some programs, however, it might not be safe to forget about failures. The answer to the next question includes a solution for these cases.

When does the observer hold the supplier accountable? Restricted asynchrony is suited for SCOOP because it alleviates the interruptibility issue and thus keeps the mechanism comprehensible. With restricted asynchrony, a client can poll implicitly or explicitly. Implicit, synchronization-oriented polling fits SCOOP well. SCOOP has three types of synchronization points: query calls, non-separate calls, and calls with lock passing. With synchronization-oriented polling, each such call triggers the client to hold its supplier accountable, making the proposed mechanism compatible. In a non-concurrent program, every feature call is non-separate; hence, every feature call becomes a poll point in the proposed mechanism, just as in the underlying mechanism. Furthermore, synchronization-oriented polling ensures that the supplier finishes issued feature calls before reporting, thus preventing race conditions (see Section 6.1). This would not be guaranteed with dispatch-oriented polling, where the client would hold the supplier accountable during asynchronous feature calls as well.

An accountability expires as soon as the supplier unlocks its request queue. With expiration of the accountability, past failures disappear as well. There are, however, some programs where failures must not disappear. For this reason, the mechanism has a *safe mode*, in which a client holds its locked suppliers accountable before their accountabilities expire. This mode is based on transaction-oriented polling. The safe mode ensures that no failure is lost, but it also reduces the potential for concurrency. In the safe mode, the client can no longer just asynchronously issue unlock requests and then continue. It first has to wait until the suppliers finished.

In summary, a client must synchronize with a supplier to learn about a failure. For long-lived suppliers, this might not be convenient for the client. For these cases, failures in suppliers can also be handled entirely in the supplier's local reaction, for example, with a callback to the client.

What is the observer's remote reaction? When the client learns about a failure of its supplier, it treats the failure as its own failure. Consequently, it assumes the role of a supplier for its own client. This ensures compatibility.

Definition 6.2 (Exception mechanism). The following instantiation of the accountability framework describes the SCOOP exception mechanism:

- *What is the supplier's local reaction?* The supplier executes its rescue clause. Without a retry instruction, the supplier first determines whether the feature call is asynchronous or synchronous. For an asynchronous call, the supplier remembers the failure and cleans up. It purges any remaining feature requests from the client, releases its obtained locks, and returns any retrieved locks. It then waits until it is held accountable or until its accountability expires. For a synchronous call, the supplier notifies the client.

- *Who is the observer?* The client is the observer.

- *When does an accountability become alive and when does it expire?* A supplier becomes accountable when its request queue gets locked; this accountability expires when the supplier unlocks its request queue. When the supplier passes its call stack lock, it creates a new accountability; this accountability expires when the supplier retrieves its call stack lock.

- *When does the observer hold the supplier accountable?* The client holds the supplier accountable as it synchronizes with the supplier.

- *What is the observer's remote reaction?* The client treats a supplier's failure as its own failure.

□

As an example, consider again the searcher program. Figure 6.3 shows an execution in which the controller calls the two searchers and passes them to the log. Meanwhile, the first searcher fails. Due to lock passing, the searchers' accountabilities persist throughout *start* and *add_entry*.

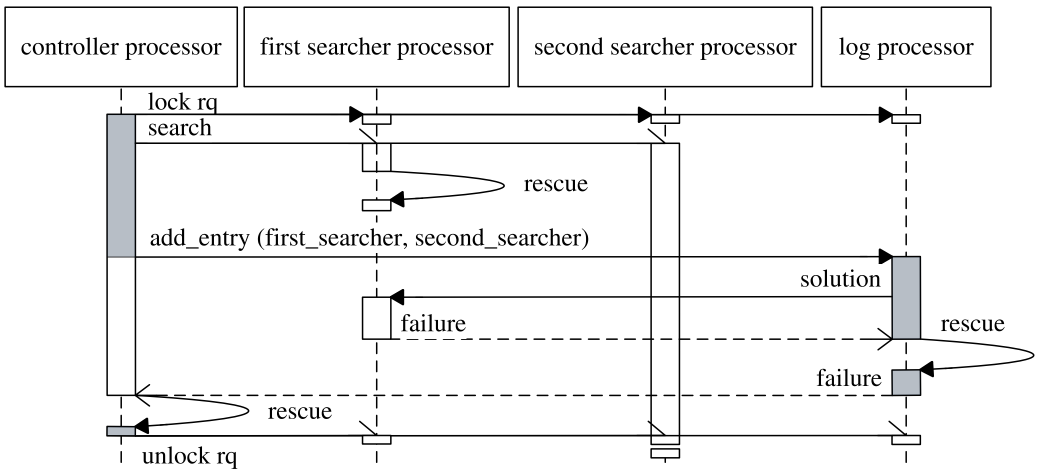


Figure 6.3: The interactions between the controller, the searchers, and the log. The shaded areas indicate the searchers' accountabilities.

The following class describes the log that writes the solutions to disk.

```
class LOG feature
  add_entry (
    first_searcher: separate SEARCHER;
    second_searcher: separate SEARCHER
  )
      -- Log one of the solutions of the searchers.
    do
      -- Has the first searcher found a solution?
      if not first_searcher.solution.is_empty then
        -- Yes. Log the first solution.
        write (first_searcher.solution)
      else
        -- No. Log the second solution.
        write (second_searcher.solution)
      end
    end
end
```

By calling *solution* synchronously, the log holds the first searcher accountable, fails, and reports the failure to the controller, who is waiting for the locks to return.

6.5 Formal specification

Support for exceptions is a cross-cutting concern because exceptions propagate through the system. This section describes the changes to the formal specification organized according to the structure in Chapter 3. It also introduces a new mechanism to support jumps between feature bodies and rescue clauses.

6.5.1 Intermediate representation

Every program of type *PROGRAM* defines settings of type *SETTINGS*. To support exceptions, *SETTINGS* has a query *safe_mode* that returns whether the program is in safe mode (see Figure 6.4). Hence *f.class_type.program.settings.safe_mode* returns whether a feature *f* is part of a program in safe mode.

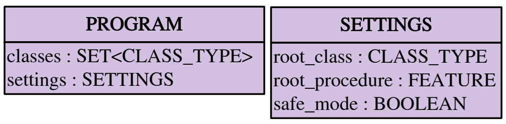


Figure 6.4: Types for the intermediate representation: failure support

6.5.2 Heap and regions

A processor *p* remembers a failure resulting from an asynchronous feature call if it cannot recover in a rescue clause. For this purpose, *REGIONS* has one more query *has_failed* that returns whether *p* has failed; the command *set_failed* sets *p*'s failure status. A new processor starts without a failure as can be seen in Figure 6.5. *REGIONS* does not keep track of accountabilities because a processor's accountability lasts exactly as long as its request queue is locked.

Just like a non-once routine, a once routine can also fail; however, a failed once routine must fail again if invoked later on. For this reason, *HEAP* must keep track of once routines that have failed. Figure 6.6 shows the changes. The state facade mirrors the changes in Figure 6.5 and Figure 6.6.

```
$\mathit{has\_failed} \colon \mathit{REGIONS} \to \mathit{PROC} \nrightarrow \mathit{BOOLEAN}$
    $k.\mathit{has\_failed}(p)$ require
        $k.\mathit{procs}.\mathit{has}(p)$
$\mathit{set\_failed} \colon \mathit{REGIONS} \to \mathit{PROC} \nrightarrow \mathit{BOOLEAN} \to \mathit{REGIONS}$
    $k.\mathit{set\_failed}(p, \mathit{hf})$ require
        $k.\mathit{procs}.\mathit{has}(p)$
    axioms
        $k.\mathit{set\_failed}(p, \mathit{hf}).\mathit{has\_failed}(p) = \mathit{hf}$
$\mathit{add\_proc} \colon \mathit{REGIONS} \to \mathit{PROC} \nrightarrow \mathit{REGIONS}$
    $k.\mathit{add\_proc}(p)$ require
        $\neg k.\mathit{procs}.\mathit{has}(p)$
    axioms
        $k.\mathit{add\_proc}(p).\mathit{procs}.\mathit{has}(p)$
        $k.\mathit{add\_proc}(p).\mathit{last\_added\_proc} = p$
        $k.\mathit{add\_proc}(p).\mathit{handled\_objs}(p).\mathit{empty}$
        $\neg k.\mathit{add\_proc}(p).\mathit{rq\_locked}(p)$
        $k.\mathit{add\_proc}(p).\mathit{cs\_locked}(p)$
        $k.\mathit{add\_proc}(p).\mathit{obtained\_rq\_locks}(p).\mathit{empty}$
        $k.\mathit{add\_proc}(p).\mathit{obtained\_cs\_lock}(p) = p$
        $k.\mathit{add\_proc}(p).\mathit{retrieved\_rq\_locks}(p).\mathit{empty}$
        $k.\mathit{add\_proc}(p).\mathit{retrieved\_cs\_locks}(p).\mathit{empty}$
        $\neg k.\mathit{add\_proc}(p).\mathit{passed}(p)$
        $\neg k.\mathit{add\_proc}(p).\mathit{has\_failed}(p)$
```

Figure 6.5: *REGIONS*: failure support

$has\_failed\colon HEAP \to PROC \to ID \nrightarrow BOOLEAN$

$\quad h.has\_failed(p, i)$ **require**

$\qquad \neg h.fresh(p, i)$

$set\_once\_func\_not\_fresh\colon HEAP \to PROC \to FEATURE \nrightarrow BOOLEAN \to BOOLEAN$
$\qquad\qquad \to REF \nrightarrow HEAP$

$\quad h.set\_once\_func\_not\_fresh(p, f, st, hf, r)$ **require**

$\qquad f \in FUNCTION \wedge f.is\_once$

$\qquad r \neq void \Rightarrow h.refs.has(r)$

axioms

$$(\exists d, c\colon \Gamma \vdash f : (d, \bullet, c)) \Rightarrow$$
$$\neg h.set\_once\_func\_not\_fresh(p, f, st, hf, r).fresh(p, f.id) \wedge$$
$$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).stable(p, f.id) = st \wedge$$
$$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).stabilizer(p, f.id) = p \wedge$$
$$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).has\_failed(p, f.id) = hf \wedge$$
$$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).once\_result(p, f.id) = \begin{cases} r & \text{if } \neg hf \\ void & \text{if } hf \end{cases}$$

$$(\exists d, c\colon \Gamma \vdash f : (d, p, c) \wedge p \neq \bullet) \Rightarrow \forall q \in PROC\colon$$
$$\neg h.set\_once\_func\_not\_fresh(p, f, st, hf, r).fresh(q, f.id) \wedge$$
$$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).stable(q, f.id) = st \wedge$$
$$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).stabilizer(q, f.id) = p \wedge$$
$$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).has\_failed(q, f.id) = hf \wedge$$
$$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).once\_result(q, f.id) = \begin{cases} r & \text{if } \neg hf \\ void & \text{if } hf \end{cases}$$

$set\_once\_proc\_not\_fresh\colon HEAP \to PROC \to FEATURE \nrightarrow BOOLEAN \to BOOLEAN$
$\qquad\qquad \to HEAP$

$\quad h.set\_once\_proc\_not\_fresh(p, f, st, hf)$ **require**

$\qquad f \in PROCEDURE \wedge f.is\_once$

axioms

$$\neg h.set\_once\_proc\_not\_fresh(p, f, st, hf).fresh(p, f.id)$$
$$h.set\_once\_proc\_not\_fresh(p, f, st, hf).stable(p, f.id) = st$$
$$h.set\_once\_proc\_not\_fresh(p, f, st, hf).stabilizer(p, f.id) = p$$
$$h.set\_once\_proc\_not\_fresh(p, f, st, hf).has\_failed(p, f.id) = hf$$

Figure 6.6: *HEAP*: failure support

6.5.3 Failure anchor mechanism

In case of a failure in a feature f, processor p aborts the feature body and executes f's rescue clause. If p executes a retry instruction, it aborts the rescue clause and retries the feature body. The failure anchor mechanism supports these statement jumps. A *failure anchor* is an operation in p's action queue providing a jump target with statements for different scenarios. Figure 6.7 shows the structure of the failure anchor. The failure anchor has a boolean flag that indicates whether a failure has occurred. Further, it has sections for different scenarios:

- The `rescue` section contains statements to rescue after a failure, provided the preceding boolean flag is true.
- The `retry` section contains statements to retry during a rescue.
- The `failure` section contains statements to fail.
- The `normal` section contains statements to be executed in case of no failure.
- The `final` section contains statements to be executed in case of no failure and to fail.

```
FAILURE_ANCHOR ≜
  failure_anchor BOOLEAN
    final STATEMENT_SEQUENCE
    normal STATEMENT_SEQUENCE
    rescue BOOLEAN STATEMENT_SEQUENCE
    retry STATEMENT_SEQUENCE
    failure STATEMENT_SEQUENCE
  end ;
FAIL ≜ fail |
  fail
    rescue BOOLEAN STATEMENT_SEQUENCE
    retry STATEMENT_SEQUENCE
    failure STATEMENT_SEQUENCE
  end ;
```

Figure 6.7: Failure anchor mechanism: operations

To fail, p calls *raise* as described in Figure 6.8.

Call runtime (raise)

$$\frac{}{\Gamma \vdash \langle p :: \texttt{call}(r_0, \mathit{raise}, ()); s_p, \sigma\rangle \to \langle p :: \texttt{fail}; s_p, \sigma\rangle}$$

Figure 6.8: Runtime calls: failure support

By performing this runtime call, p executes the `fail` operation. The `fail` operation has optional sections to overwrite the `rescue`, `retry`, and `failure` sections of the next failure anchor in p's action queue. Figure 6.9 shows the transition rules: to fail with overwrite, p skips the statements $s_1; s_n$ to the next failure anchor, sets the failure flag to true, overwrites the sections, and then continues with the failure anchor; to fail without overwrite, p only changes the failure flag.

Processor p then executes the failure anchor, as described in Figure 6.10. If the rescue flag is set, p sets up a new failure anchor where the rescue flag is not set and executes the `rescue` statements; the new failure anchor provides a jump target for a retry instruction, as described in Figure 6.11. In case p encounters a retry instruction, p replaces the remaining statements with the `retry` statements of the new failure anchor; it also sets the failure flag of the new failure anchor to false, providing a jump target for new failures. Without a retry instruction, p reaches the new failure anchor but does not execute the `rescue` statements again; instead p executes the `final` statements followed by the `failure` statements.

If p reaches a failure anchor without a failure, it executes the `final` statements followed by the `normal` statements (see Figure 6.10).

Fail (without failure anchor overwrite)

$$\frac{\forall i \in 1,\dots,n\colon \neg s_i \in \mathit{FAILURE\_ANCHOR}}{\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{fail}; \\
\qquad s_1;\dots;s_n; \\
\qquad \texttt{failure\_anchor}\ \mathit{false} \\
\qquad\quad \texttt{final}\ s_{\mathit{final}} \\
\qquad\quad \texttt{normal}\ s_{\mathit{normal}} \\
\qquad\quad \texttt{rescue}\ \mathit{rescue}\ s_{\mathit{rescue}} \\
\qquad\quad \texttt{retry}\ s_{\mathit{retry}} \\
\qquad\quad \texttt{failure}\ s_{s_{\mathit{failure}}} \\
\qquad \texttt{end}; \\
\qquad s_p, \sigma\rangle \rightarrow \\
\quad \langle p :: \texttt{fail}\ \texttt{rescue}\ \mathit{rescue}\ s_{\mathit{rescue}}\ \texttt{retry}\ s_{\mathit{retry}}\ \texttt{failure}\ s_{\mathit{failure}}\ \texttt{end}; \\
\qquad s_p, \sigma\rangle
\end{array}}$$

Fail (with failure anchor overwrite)

$$\frac{\forall i \in 1,\dots,n\colon \neg s_i \in \mathit{FAILURE\_ANCHOR}}{\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{fail}\ \texttt{rescue}\ \mathit{rescue}'\ s'_{\mathit{rescue}}\ \texttt{retry}\ s'_{\mathit{retry}}\ \texttt{failure}\ s'_{\mathit{failure}}\ \texttt{end}; \\
\qquad s_1;\dots;s_n; \\
\qquad \texttt{failure\_anchor}\ \mathit{false} \\
\qquad\quad \texttt{final}\ s_{\mathit{final}} \\
\qquad\quad \texttt{normal}\ s_{\mathit{normal}} \\
\qquad\quad \texttt{rescue}\ \mathit{rescue}\ s_{\mathit{rescue}} \\
\qquad\quad \texttt{retry}\ s_{\mathit{retry}} \\
\qquad\quad \texttt{failure}\ s_{s_{\mathit{failure}}} \\
\qquad \texttt{end}; \\
\qquad s_p, \sigma\rangle \rightarrow \\
\quad \langle p :: \texttt{failure\_anchor}\ \mathit{true} \\
\qquad\quad \texttt{final}\ s_{\mathit{final}} \\
\qquad\quad \texttt{normal}\ s_{\mathit{normal}} \\
\qquad\quad \texttt{rescue}\ \mathit{rescue}'\ s'_{\mathit{rescue}} \\
\qquad\quad \texttt{retry}\ s'_{\mathit{retry}} \\
\qquad\quad \texttt{failure}\ s'_{\mathit{failure}} \\
\qquad \texttt{end}; \\
\qquad s_p, \sigma\rangle
\end{array}}$$

Figure 6.9: Failure anchor mechanism: fail transition rule

Failure anchor

```
________________________________________
$\Gamma \vdash \langle p$ :: failure_anchor $failure$
            final $s_{final}$
            normal $s_{normal}$
            rescue $rescue$ $s_{rescue}$
            retry $s_{retry}$
            failure $s_{failure}$
         end;
         $s_p, \sigma\rangle \longrightarrow$
    $\langle p$ :: provided $\neg failure$ then
            $s_{final}$; $s_{normal}$
         else
            provided $rescue$ then
               $s_{rescue}$;
               failure_anchor $failure$
                  final $s_{final}$
                  normal $s_{normal}$
                  rescue $false$ $s_{rescue}$
                  retry $s_{retry}$
                  failure $s_{failure}$
               end
            else
               $s_{final}$; $s_{failure}$
            end
         end;
         $s_p, \sigma\rangle$
```

Figure 6.10: Failure anchor mechanism: failure anchor transition rule

Retry instruction

$$\frac{\forall i \in \{1,\ldots,n\}\colon \neg s_i \in \mathit{FAILURE\_ANCHOR}}{\begin{array}{l}
\Gamma \vdash \langle p :: \mathbf{retry};\\
\qquad\qquad s_1; \ldots; s_n;\\
\qquad\qquad \texttt{failure\_anchor}\ \mathit{true}\\
\qquad\qquad\quad \texttt{final}\ s_{\mathit{final}}\\
\qquad\qquad\quad \texttt{normal}\ s_{\mathit{normal}}\\
\qquad\qquad\quad \texttt{rescue}\ \mathit{false}\ s_{\mathit{rescue}}\\
\qquad\qquad\quad \texttt{retry}\ s_{\mathit{retry}}\\
\qquad\qquad\quad \texttt{failure}\ s_{s_{\mathit{failure}}}\\
\qquad\qquad \texttt{end};\\
\qquad\qquad s_p, \sigma\rangle \rightarrow\\
\qquad \langle p :: s_{\mathit{retry}};\\
\qquad\qquad \texttt{failure\_anchor}\ \mathit{false}\\
\qquad\qquad\quad \texttt{final}\ s_{\mathit{final}}\\
\qquad\qquad\quad \texttt{normal}\ s_{\mathit{normal}}\\
\qquad\qquad\quad \texttt{rescue}\ \mathit{rescue}\ \mathit{true}\\
\qquad\qquad\quad \texttt{retry}\ s_{\mathit{retry}}\\
\qquad\qquad\quad \texttt{failure}\ s_{\mathit{failure}}\\
\qquad\qquad \texttt{end};\\
\qquad\qquad s_p, \sigma\rangle
\end{array}}$$

Figure 6.11: Failure anchor mechanism: retry instruction transition rule

6.5.4 Notification mechanism

To support synchronous failure notifications, the `notify` operation receives a new flag that indicates failures, as shown in Figure 6.12. A supplier q executes $\texttt{notify}(a, \mathit{true})$ to signal a failure to a client p waiting on channel a; p then propagates the failure. To notify an asynchronous client p, q cannot use $\texttt{notify}(a, \mathit{true})$ because p is not waiting on a. Instead, q calls $\sigma.\mathit{set\_failed}(q)$ (see Section 6.5.2); p can then check $\sigma.\mathit{has\_failed}(q)$ next time it waits for q and propagate the failure. To support this, the `wait` operation receives a new argument for the processor to wait for, as shown in Figure 6.12.

$\mathit{NOTIFY} \triangleq \texttt{notify}\ \ \mathit{TUPLE}\langle \mathit{CHANNEL},\ \mathit{BOOLEAN}\rangle$;
$\mathit{WAIT} \triangleq \texttt{wait}\ \ \mathit{TUPLE}\langle \mathit{CHANNEL},\ \mathit{PROC}\rangle$;

Wait for result (non-separate)

$$\frac{}{\Gamma \vdash \langle p :: \texttt{result}(a, r); s_w; \texttt{wait}(a, p); s_p, \sigma\rangle \to \langle p :: s_w; s_p[r/a.\mathit{data}], \sigma\rangle}$$

Wait for notify (non-separate)

$$\frac{}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{notify}(a, \mathit{hf}); s_w; \texttt{wait}(a, p); s_p, \sigma\rangle \to \\ \quad \langle p :: s_w; \texttt{provided}\ \mathit{hf}\ \texttt{then fail else nop end}; s_p, \sigma\rangle\end{array}}$$

Wait for result (separate)

$$\frac{}{\Gamma \vdash \langle p :: s_w; \texttt{wait}(a, q); s_p \mid q :: \texttt{result}(a, r); s_q, \sigma\rangle \to \langle p :: s_w; s_p[r/a.\mathit{data}] \mid q :: s_q, \sigma\rangle}$$

Wait for notify (separate)

$$\frac{}{\begin{array}{l}\Gamma \vdash \langle p :: s_w; \texttt{wait}(a, q); s_p \mid q :: \texttt{notify}(a, \mathit{hf}); s_q, \sigma\rangle \to \\ \quad \langle p :: s_w; \texttt{provided}\ \mathit{hf}\ \texttt{then fail else nop end}; s_p \mid q :: s_q, \sigma\rangle\end{array}}$$

Wait for failure (separate)

$$\frac{\sigma.\mathit{has\_failed}(q)}{\Gamma \vdash \langle p :: \texttt{wait}(a, q); s_p \mid q :: s_q, \sigma\rangle \to \langle p :: \texttt{fail}; s_p \mid q :: s_q, \sigma\rangle}$$

Figure 6.12: Notification mechanism: failure support

The changes to `notify` and `wait` trigger a number of small adaptations: the transition rules for feature calls, flow control instructions, and assignment instructions must use the extended signatures.

6.5.5 Locking and unlocking mechanism

An accountability of a supplier p expires when p unlocks its request queue using `unlock_rq`; hence, p can forget about any failures at that point. Figure 6.13 extends the `unlock_rq` operation accordingly. Once p's request queue gets locked anew, a new accountability starts.

UNLOCK REQUEST QUEUE

$$\frac{\begin{array}{l}\sigma.rq\_locked(p) \\ \forall q \in \sigma.procs\colon \neg\sigma.rq\_locks(q).has(p) \\ \sigma' \overset{def}{=} \sigma.unlock\_rq(p).set\_failed(p, false)\end{array}}{\Gamma \vdash \langle p :: \texttt{unlock\_rq}; s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Figure 6.13: Locking and unlocking mechanism: failure support

6.5.6 Feature applications

A supplier p can encounter a failure while applying a feature f. Hence, it can either return successfully or with a failure. For this purpose, the `return` operation receives a new flag that indicates whether a failure has occurred, as shown in Figure 6.14 for a successful return and in Figure 6.15 for a return with a failure. For a successful return, p sets f to not fresh, stable, and not failed if the flag snf (see Section 5.3) is set; for a failed return, p sets f to not fresh, stable, and failed. In addition to snf, the `return` has a new flag sf to indicate whether p should set f as fresh if f is a once routine; this support is necessary to handle failures in preconditions. For a failed return, p determines the type of call from the client q: if the call is non-separate, p fails; if the call is separate and synchronous, p notifies q of the failure; if the call is separate and asynchronous, p purges any remaining feature requests using the `purge` operation described in Figure 6.16. During a purge, p discards any feature requests until its request queue lock gets released, at which point p can forget about its failure and accept new feature requests.

$$
\begin{aligned}
&\mathit{RETURN} \triangleq\\
&\quad \texttt{return}\ \mathit{TUPLE}\langle \mathit{CHANNEL},\ \mathit{FEATURE},\ \mathit{PROC},\\
&\qquad \mathit{TUPLE}\langle \mathit{SET}\langle \mathit{PROC}\rangle,\ \mathit{SET}\langle \mathit{PROC}\rangle\rangle,\ \mathit{BOOLEAN},\ \mathit{BOOLEAN},\ \mathit{BOOLEAN}\rangle\ \mid\\
&\quad \texttt{return}\ \mathit{TUPLE}\langle \mathit{CHANNEL},\ \mathit{FEATURE},\ \mathit{REF},\ \mathit{PROC},\\
&\qquad \mathit{TUPLE}\langle \mathit{SET}\langle \mathit{PROC}\rangle,\ \mathit{SET}\langle \mathit{PROC}\rangle\rangle,\ \mathit{BOOLEAN},\ \mathit{BOOLEAN},\ \mathit{BOOLEAN}\rangle\ ;
\end{aligned}
$$

Return (successful command)

$$
\dfrac{
\begin{array}{l}
\sigma' \overset{def}{=} \begin{cases} \sigma.\mathit{set\_once\_proc\_not\_fresh}(p, f, \mathit{true}, \mathit{false}) & \text{if } f.\mathit{is\_once} \wedge \mathit{snf}\\ \sigma.\mathit{set\_once\_rout\_fresh}(p, f) & \text{if } f.\mathit{is\_once} \wedge \mathit{sf}\\ \sigma & \text{otherwise}\end{cases}\\
\sigma'' \overset{def}{=} \sigma'.\mathit{pop\_env}(p).\mathit{revoke\_locks}(q, p)
\end{array}
}{
\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{return}(a, f, q, (\bar{l}_r, \bar{l}_c), \mathit{false}, \mathit{snf}, \mathit{sf}); s_p, \sigma\rangle \rightarrow\\
\quad \langle p :: \texttt{provided}\ (\bar{l}_r, \bar{l}_c) \neq (\{\}, \{\})\ \texttt{then}\ \texttt{notify}(a, \mathit{false})\ \texttt{else}\ \texttt{nop}\ \texttt{end};\\
\qquad s_p, \sigma''\rangle
\end{array}
}
$$

Return (successful query)

$$
\dfrac{
\begin{array}{l}
(\sigma', r_r') \overset{def}{=} \begin{cases} \text{if } r_r \neq \mathit{void} \wedge \sigma.\mathit{ref\_obj}(r_r).\mathit{class\_type}.\mathit{is\_exp} \wedge \sigma.\mathit{handler}(r_r) \neq q\\ \quad (\sigma^*, \sigma^*.\mathit{last\_imported\_obj})\\ \quad\ \ \textbf{where}\\ \quad\quad \sigma^* \overset{def}{=} \sigma.\mathit{import}(q, r_r)\\ \text{otherwise}\\ \quad (\sigma, r_r)\end{cases}\\
\sigma'' \overset{def}{=} \begin{cases} \sigma'.\mathit{set\_once\_func\_not\_fresh}(p, f, \mathit{true}, \mathit{false}, r_r) & \text{if } f.\mathit{is\_once} \wedge \mathit{snf}\\ \sigma'.\mathit{set\_once\_rout\_fresh}(p, f) & \text{if } f.\mathit{is\_once} \wedge \mathit{sf}\\ \sigma' & \text{otherwise}\end{cases}\\
\sigma''' \overset{def}{=} \sigma''.\mathit{pop\_env}(p).\mathit{revoke\_locks}(q, p)
\end{array}
}{
\Gamma \vdash \langle p :: \texttt{return}(a, f, r_r, q, (\bar{l}_r, \bar{l}_c), \mathit{false}, \mathit{snf}, \mathit{sf}); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a, r_r'); s_p, \sigma'''\rangle
}
$$

Figure 6.14: Return: success

Return (failure)

$$\sigma' \overset{def}{=} \begin{cases} \text{if } f \in \mathit{FUNCTION} \land f.\mathit{is\_once} \land \mathit{snf} \\ \quad \sigma.\mathit{set\_once\_func\_not\_fresh}(p, f, \mathit{true}, \mathit{true}, \mathit{void}) \\ \text{if } f \in \mathit{PROCEDURE} \land f.\mathit{is\_once} \land \mathit{snf} \\ \quad \sigma.\mathit{set\_once\_proc\_not\_fresh}(p, f, \mathit{true}, \mathit{true}) \\ \text{if } f.\mathit{is\_once} \land \mathit{sf} \\ \quad \sigma.\mathit{set\_once\_rout\_fresh}(p, f) \\ \text{otherwise} \\ \quad \sigma \end{cases}$$

$$\sigma'' \overset{def}{=} \begin{cases} \sigma'.\mathit{set\_failed}(p, \mathit{true}) & \text{if } p \neq q \land f \notin \mathit{FUNCTION} \land (\bar{l}_r, \bar{l}_c) = (\{\}, \{\}) \\ \sigma' & \text{otherwise} \end{cases}$$

$$\sigma''' \overset{def}{=} \sigma''.\mathit{pop\_env}(p).\mathit{revoke\_locks}(q, p)$$

```
Γ ⊢⟨$p$ :: return($a, f, q, (\bar{l}_r, \bar{l}_c), \mathit{true}, \mathit{snf}, \mathit{sf}$); $s_p, \sigma$⟩ →
   ⟨$p$ :: provided $p = q$ then fail else
            provided $f \notin \mathit{FUNCTION} \land (\bar{l}_r, \bar{l}_c) = (\{\}, \{\})$ then
               purge
            else
               notify($a, \mathit{true}$)
            end
         end;
         $s_p, \sigma'''$⟩
```

Figure 6.15: Return: failure

$\mathit{PURGE} \triangleq$ purge ;

Purge request queue unlock request

$\Gamma \vdash \langle p :: \texttt{purge}; \texttt{unlock\_rq}; s_p, \sigma\rangle \rightarrow \langle p :: \texttt{unlock\_rq}; s_p, \sigma\rangle$

Purge feature request

$\Gamma \vdash \langle p :: \texttt{purge}; \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{purge}; s_p, \sigma\rangle$

Purge notification request

$\Gamma \vdash \langle p :: \texttt{purge}; \texttt{notify}(a, \mathit{false}); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{purge}; s_p, \sigma\rangle$

Figure 6.16: Purging

To apply a non-once routine or a fresh once routine f, p sets f to not fresh, not stable, and not failed if f is a once routine. It then sets up a failure anchor as shown in Figure 6.18. Without a failure, p releases the locks and returns successfully, setting f to stable and not failed if f is a once routine. In case of a failure, p executes the rescue clause, as described in Figure 6.17; in case f is a once function, p updates f's once status as not fresh, not stable, and not failed whenever it writes to the result entity. In case of a retry, p restores the feature body. Without a retry, p releases the locks and returns with a failure, setting f to stable and failed if f is a once routine. After setting up the failure anchor, p obtains the necessary locks and checks the precondition. Once the precondition is satisfied, p executes the body of f. It then evaluates the postcondition and the invariant.

$EXECUTE\_RESCUE\_CLAUSE \triangleq \texttt{execute\_rescue\_clause}\ \ TUPLE\langle FEATURE\rangle$;

EXECUTE RESCUE CLAUSE

$$\frac{}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{execute\_rescue\_clause}(f);\ s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ f \in FUNCTION \wedge f.is\_once\ \texttt{then} \\ \qquad\qquad f.rescue\_clause \\ \qquad\qquad\quad [result := y;\ \texttt{set\_not\_fresh}(f)/result := y] \\ \qquad\qquad\quad [\textbf{create}\ result.y;\ \texttt{set\_not\_fresh}(f)/\textbf{create}\ result.y] \\ \qquad\quad \texttt{else} \\ \qquad\qquad f.rescue\_clause \\ \qquad\quad \texttt{end}; \\ \qquad\quad s_p, \sigma\rangle\end{array}}$$

SET NOT FRESH

$$\frac{\begin{array}{l}f \in FUNCTION \wedge f.is\_once \\ \sigma.envs(p).top.names.has(result.name) \\ \sigma' \overset{def}{=} \sigma.set\_once\_func\_not\_fresh(p, f, false, false, \sigma.val(p, result.name))\end{array}}{\Gamma \vdash \langle p :: \texttt{set\_not\_fresh}(f);\ s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma\rangle}$$

Figure 6.17: Execute rescue clause

Apply feature (non-once routine or fresh once routine)

$$f \in \mathit{ROUTINE} \wedge (f.\mathit{is\_once} \Rightarrow \sigma.\mathit{fresh}(p, f.\mathit{id}))$$
$$\sigma.\mathit{handler}(r_0) = p$$
$$\sigma' \overset{def}{=} \begin{cases} \text{if } f \in \mathit{FUNCTION} \wedge f.\mathit{is\_once} \\ \quad \sigma.\mathit{set\_once\_func\_not\_fresh}(p, f, \mathit{false}, \mathit{false}, \mathit{void}) \\ \text{if } f \in \mathit{PROCEDURE} \wedge f.\mathit{is\_once} \\ \quad \sigma.\mathit{set\_once\_proc\_not\_fresh}(p, f, \mathit{false}, \mathit{false}) \\ \text{otherwise} \quad \sigma \end{cases}$$
$$\sigma'' \overset{def}{=} \sigma'.\mathit{pass\_locks}(q, p, (\bar{l}_r, \bar{l}_c)).\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n))$$
$$\bar{g}_{\mathit{required\_rq\_and\_cs\_locks}} \overset{def}{=} \{p\} \cup \{x \in \mathit{PROC} \mid \exists i \in \{1, \ldots, n\}, g, c \colon \Gamma \vdash f.\mathit{formals}(i) : (!, g, c) \wedge \sigma''.\mathit{ref\_obj}(r_i).\mathit{class\_type}.\mathit{is\_ref} \wedge x = \sigma''.\mathit{handler}(r_i)\}$$
$$\bar{g}_{\mathit{required\_cs\_locks}} \overset{def}{=} \{x \in \bar{g}_{\mathit{required\_rq\_and\_cs\_locks}} \mid x = p \vee (x \neq p \wedge \sigma''.\mathit{passed}(x) \wedge \neg \sigma''.\mathit{passed}(p) \wedge \sigma''.\mathit{cs\_locks}(p).\mathit{has}(x)\}$$
$$\bar{g}_{\mathit{required\_rq\_locks}} \overset{def}{=} \bar{g}_{\mathit{required\_rq\_and\_cs\_locks}} \setminus \bar{g}_{\mathit{required\_cs\_locks}}$$
$$\{g_1, \ldots, g_m\} \overset{def}{=} \bar{g}_{\mathit{missing\_rq\_locks}} \overset{def}{=} \{x \in \bar{g}_{\mathit{required\_rq\_locks}} \mid \neg \sigma''.\mathit{rq\_locks}(p).\mathit{has}(x)\}$$
a' is fresh

```
$\Gamma \vdash \langle p ::$ apply$(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow$
  $\langle p ::$ check_pre_and_lock$(a, f, q, (\bar{l}_r, \bar{l}_c), \bar{g}_{missing\_rq\_locks})$;
      execute_body$(f, \bar{g}_{missing\_rq\_locks})$; check_post_and_inv$(f)$;
      failure_anchor $\mathit{false}$
         final
            issue$(g_1,$ unlock_rq$)$; $\ldots$ ; issue$(g_m,$ unlock_rq$)$;
            pop_obtained_locks
         normal
            provided $f \in \mathit{FUNCTION}$ then
               read$(\mathit{result}, a')$; return$(a, f, a'.\mathit{data}, q, (\bar{l}_r, \bar{l}_c), \mathit{false}, \mathit{true}, \mathit{false})$
            else
               return$(a, f, q, (\bar{l}_r, \bar{l}_c), \mathit{false}, \mathit{true}, \mathit{false})$
            end
         rescue $\mathit{true}$ execute_rescue_clause$(f)$
         retry execute_body$(f, \bar{g}_{missing\_rq\_locks})$; check_post_and_inv$(f)$
         failure return$(a, f, q, (\bar{l}_r, \bar{l}_c), \mathit{true}, \mathit{true}, \mathit{false})$
      end; $s_p, \sigma''\rangle$
```

Figure 6.18: Non-once or fresh once routine application: failure support

A precondition is either a correctness condition or a wait condition. If violated, a correctness condition remains violated because the state of its feature call targets cannot change; since it is the client's responsibility to establish the precondition, the client fails. A wait condition, on the other hand, involves targets that can change between evaluations; the supplier waits until the precondition is satisfied. According to the informal description, a correctness condition only involves feature calls on targets that are controlled in the calling context. This static property does not precisely cover all scenarios where the correctness condition cannot change. For example, consider a client obtaining a request queue lock on a target at the beginning of a non-separate call sequence; in the last non-separate call, it synchronously calls the supplier and passes the target as an argument. The supplier has a failed precondition on the target; it determines that the precondition is a wait condition because the target is not controlled. However, since the target is locked and the client is waiting, the precondition will not change. A dynamic property is more precise: a precondition of a feature is a correctness condition if the feature call is synchronous and the client has a lock on each target. This property implies that all targets remain unchanged because both the client and the supplier are waiting, and no other processor can access the targets; hence the precondition remains unchanged.

Clarification 6.1 (Preconditions). A precondition of a feature f is a correctness condition if and only if (1) the call to f is synchronous and (2) the client claims the request queue lock or the call stack lock for every target in the precondition. Every other precondition is a wait condition. □

Figure 6.19 shows the revised transition rule to check a precondition. The supplier p first determines the handlers $\overline{g}$ of the precondition targets: for each target, it determines the controlling entity and finds the handler of that entity; p itself is not part of this set. It then determines whether the precondition is a correctness condition (see cc), i.e., (1) the call is synchronous because it is non-separate ($p = q$), it is a function call ($f \in FUNCTION$), or it involves lock passing $((\overline{l}_r, \overline{l}_c) \neq (\{\}, \{\}))$, and (2) the client q claims a lock for every target ($\overline{g} \subseteq \sigma.rq\_locks(q) \cup \sigma.cs\_locks(q)$). If p finds a correctness condition unsatisfied, it fails to propagate the failure to the client. Because p is not responsible for the failure, it must not execute a rescue clause, and it must set f to fresh again if f is a once routine. To do so, it overwrites the failure anchor accordingly. If p finds a wait condition unsatisfied, it releases the locks and tries again.

$CHECK\_PRE\_AND\_LOCK \triangleq$
`check_pre_and_lock` $TUPLE\langle CHANNEL,\ FEATURE,\ PROC,$
$TUPLE\langle SET\langle PROC\rangle,\ SET\langle PROC\rangle\rangle,\ SET\langle PROC\rangle\rangle$;

Check precondition and lock

$$targets(e) \overset{def}{=} \begin{cases} \{e_0\} \bigcup_{i=1,\ldots,n} targets(e_i) & \text{if } e = e_0.h(e_1,\ldots,e_n) \\ \{\} & \text{otherwise} \end{cases}$$

$\overline{g} \overset{def}{=} \{x \in \sigma.procs \mid (\exists y, z \in EXPRESSION\colon\ y \in targets(f.pre)$
$\quad \wedge z = controlling\_entity(\Gamma, y)) \wedge x = \sigma.handler(\sigma.val(p, z.name)) \wedge x \neq p\}$

a is fresh

$cc \overset{def}{=} (p = q \vee f \in FUNCTION \vee (\bar{l}_r, \bar{l}_c) \neq (\{\},\{\})\wedge$
$\quad (\overline{g} \subseteq \sigma.rq\_locks(q) \cup \sigma.cs\_locks(q))$

```
Γ ⊢⟨p :: check_pre_and_lock(a, f, q, (l̄_r, l̄_c), {g_1, ..., g_m}); s_p, σ⟩ →
  ⟨p :: lock({g_1, ..., g_m});
      provided f.has_pre then
        eval(a, f.pre);
        wait(a, p);
        provided a.data then
          nop
        else
          provided cc then
            fail
              rescue false nop
              retry nop
              failure return(a, f, q, (l̄_r, l̄_c), true, false, true)
            end
          else
            issue(g_1, unlock_rq);
            ...
            issue(g_m, unlock_rq);
            pop_obtained_locks;
            check_pre_and_lock(a, f, q, (l̄_r, l̄_c), {g_1, ..., g_m})
          end
        end
      else
        nop
      end;
      s_p, σ⟩
```

Figure 6.19: Check precondition and lock: failure support

Figure 6.20 shows the transition rule to execute the body of f. If f is a once function, then p sets the once result as not fresh, not stable, and not failed whenever it writes to the result entity. To support the safe mode, `execute_body` receives the set of processors $\{g_1, \ldots, g_n\}$ whose request queues p is going to release. In case p executes a program in safe mode, it asks each processor in $\{g_1, \ldots, g_n\}$ to send a notification after finishing the issued workload; p then waits for these processors. In case none of them fails, p receives a notification from each of them and continues; in case a processor $g \in \{g_1, \ldots, g_n\}$ fails asynchronously, g records its failure in the state (see Figure 6.15) and purges the notification request along with all further feature requests (see Figure 6.16); p then propagates the failure (see Figure 6.12).

$EXECUTE\_BODY \triangleq$ `execute_body` $TUPLE\langle FEATURE,\ SET\langle PROC\rangle\rangle$;

Execute body

$$\frac{\forall i \in \{1, \ldots, n\}\colon a_i \text{ is fresh}}{\Gamma \vdash \langle p :: \mathtt{execute\_body}(f, \{g_1, \ldots, g_n\}); s_p, \sigma\rangle \rightarrow}$$

```
    ⟨p :: provided $f \in FUNCTION \wedge f.is\_once$ then
            $f.body$
              [$result := y$; set_not_fresh($f$)/$result := y$]
              [create $result.y$; set_not_fresh($f$)/create $result.y$]
        else
            $f.body$
        end;
        provided $f.class\_type.program.settings.safe\_mode$ then
            issue($g_1$, notify($a_1$, $false$)); ... ; issue($g_n$, notify($a_n$, $false$));
            wait($a_1$, $g_1$); ... ; wait($a_n$, $g_n$)
        else
            nop
        end;
        $s_p, \sigma$⟩
```

Figure 6.20: Execute body: failure support

After the body execution, p evaluates the postcondition and the invariant, as shown in Figure 6.21. If p finds one of them not to hold, it fails; this time, p is responsible for the contract violation.

CHECK POSTCONDITION AND INVARIANT

$$\frac{\begin{array}{c} a_{inv} \text{ is fresh} \\ a_{post} \text{ is fresh} \end{array}}{\Gamma \vdash \langle p :: \texttt{check\_post\_and\_inv}(f); s_p, \sigma\rangle \rightarrow}$$

```
⟨p :: provided $f.has\_post$ then
         eval($a_{post}$, $f.post$); wait($a_{post}$, $p$);
         provided $a_{post}.data$ then nop else fail end
     else
         nop
     end;
     provided $f.class\_type.has\_inv \wedge f.is\_exported$ then
         eval($a_{inv}$, $f.class\_type.inv$); wait($a_{inv}$, $p$);
         provided $a_{inv}.data$ then nop else fail end
     else
         nop
     end; $s_p, \sigma$⟩
```

Figure 6.21: Check postcondition and invariant: failure support

To apply a once routine f that is not fresh, p sets up a failure anchor, as can be seen in Figure 6.23. In case f has failed, p does not execute a rescue clause but simply fails again without changing the status of f; in case of no failure, p returns as before, not changing the status of f. To apply an attribute, p behaves as before.

APPLY FEATURE (ATTRIBUTE)

$$\frac{\begin{array}{l} f \in ATTRIBUTE \\ \sigma.handler(r_0) = p \\ \sigma' \overset{def}{=} \sigma.pass\_locks(q, p, (\bar{l}_r, \bar{l}_c)).push\_env\_with\_feature(p, f, r_0, ()) \\ a' \text{ is fresh} \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a', f); \texttt{wait}(a', p); \\ \qquad \texttt{return}(a, f, a'.data, q, (\bar{l}_r, \bar{l}_c), false, false, false); s_p, \sigma'\rangle \end{array}}$$

Figure 6.22: Attribute application: failure support

Apply feature (not fresh once routine)

$$\frac{\begin{array}{l} f \in \mathit{ROUTINE} \wedge f.\mathit{is\_once} \wedge \\ \quad \neg \sigma.\mathit{fresh}(p, f.\mathit{id}) \wedge (\sigma.\mathit{stable}(p, f.\mathit{id}) \vee \sigma.\mathit{stabilizer}(p, f.\mathit{id}) = p) \\ \sigma.\mathit{handler}(r_0) = p \\ \sigma' \overset{def}{=} \sigma.\mathit{pass\_locks}(q, p, (\bar{l}_r, \bar{l}_c)).\mathit{push\_env\_with\_feature}(p, f, r_0, (r_1, \ldots, r_n)) \end{array}}{\Gamma \vdash \langle p :: \mathtt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma \rangle \rightarrow}$$

```
⟨p :: provided $\sigma'.\mathit{has\_failed}(p, f)$ then
          fail
     else
          nop
     end;
     failure_anchor $\mathit{false}$
          final nop
          normal
               provided $f \in \mathit{FUNCTION}$ then
                    return($a, f, \sigma'.\mathit{once\_result}(p, f.\mathit{id}), q, (\bar{l}_r, \bar{l}_c), \mathit{false}, \mathit{false}, \mathit{false}$)
               else
                    return($a, f, q, (\bar{l}_r, \bar{l}_c), \mathit{false}, \mathit{false}, \mathit{false}$)
               end
          rescue $\mathit{false}$ nop
          retry nop
          failure return($a, f, q, (\bar{l}_r, \bar{l}_c), \mathit{true}, \mathit{false}, \mathit{false}$)
     end;
     $s_p, \sigma'$⟩
```

Figure 6.23: Not fresh once routine application: failure support

6.6 Testing

The test suite used in Chapter 5 also includes tests for the exception mechanism. These tests revealed two flaws concerning the safe mode in the initial design.

6.6.1 *Safe mode order*

Before releasing the request queue locks of $\{g_1, \ldots, g_n\}$ in safe mode, processor p asks each $g \in \{g_1, \ldots, g_n\}$ to send a notification after finishing the issued workload; p then waits for these processors.

Flaw

The order in which p asks for notifications is problematic. Consider the following:

```
class A create make feature
  make
    local
      b1: separate B
      b2: separate B
    do
      create b1.make; create b2.make
      f (b1, b2)
    end

  f (b1: separate B; b2: separate B)
    do
      b1.g
      b2.h
    end
end

class B create make feature
  make
    do end

  g
    do raise end

  h
    do end
end
```

Using Maude's **rew** command, a new root processor p_1 creates two objects on two new processors p_3 and p_5. It starts executing the feature f and locks the two request queues. It then asynchronously calls both of them; p_5 succeeds, but p_3 fails, purging any further feature requests and recording its failure in the state. In the meantime, p_1 issues notification requests to both processors and waits for their report:

$\langle$

$p_1 :: \texttt{wait}(a_3, p_3); \texttt{wait}(a_5, p_5); \ldots \mid$
$p_3 :: \texttt{purge}; \texttt{notify}(a_3, \mathit{true}) \mid$
$p_5 :: \texttt{notify}(a_5, \mathit{true})$

,

locks:
$p_1 ::$ orq: $(\{\}, \{p_3, p_5\})$ rrq: $(\{\}, \{\})$ rcs: $(\{\}, \{\})$ locked
$p_3 ::$ orq: $()$ rrq: $()$ rcs: $()$ locked
$p_5 ::$ orq: $()$ rrq: $()$ rcs: $()$ locked
failures:
p_3

$\rangle$

Since p_3 failed, p_1 fails as well without receiving p_5's notification, causing p_5 to wait indefinitely.

Clarification

Before releasing the request queue locks of $\{g_1, \ldots, g_n\}$ in safe mode, processor p must wait for the notification of each $g \in \{g_1, \ldots, g_n\}$ before asking the next one for a notification, as shown in Figure 6.24. This change prevents any pending notifications.

Clarification 6.2 (Failures). Before releasing the request queue locks of processors $\{g_1, \ldots, g_n\}$ in safe mode, a processor p asks each $g \in \{g_1, \ldots, g_n\}$ to send a notification after finishing the issued workload; for all $i \in \{1, n-1\}$, p waits for g_i's notification before asking g_{i+1}. $\square$

```
EXECUTE BODY
                    $\forall i \in \{1, \ldots, n\}\colon a_i$ is fresh
-----------------------------------------------------------------
$\Gamma \vdash \langle p ::$ execute_body$(f, \{g_1, \ldots, g_n\}); s_p, \sigma\rangle \rightarrow$
    $\langle p ::$ provided $f \in FUNCTION \wedge f.is\_once$ then
            $f.body$
                $[result := y;$ set_not_fresh$(f) / result := y]$
                $[$create $result.y;$ set_not_fresh$(f) /$ create $result.y]$
        else
            $f.body$
        end;
        provided $f.class\_type.program.settings.safe\_mode$ then
            issue$(g_1,$ notify$(a_1, false));$ wait$(a_1, g_1);$
            $\ldots;$
            issue$(g_n,$ notify$(a_n, false));$ wait$(a_n, g_n)$
        else
            nop
        end;
        $s_p, \sigma\rangle$
```

Figure 6.24: Safe mode clarification: body execution

6.6.2 Safe mode and creation procedures

The safe mode should guarantee that no supplier fails without notifying its client. For this reason, every client p executes a check point after executing a feature body (see Figure 6.20 and Figure 6.24) to wait for suppliers before releasing their request queue locks.

Flaw

The current check point does not guarantee that all failures propagate to the client. Consider the following program:

```
class A create make feature
  make
    local
      b: separate B
    do
      create b.make
    end
end

class B create make feature
  make
    do
      raise
    end
end
```

Using Maude's **rew** command, a new root processor p_1 starts executing *make*. In *make*, p_1 executes the creation instruction (see Figure 3.53). It creates a new object on a new processors p_3 . It then locks the request queue of p_3 and issues an asynchronous feature call to the failing creation routine. It then releases the request queue lock; consequently, p_3 forgets about its failure. Ending *make*, p_1 executes the safe mode check point. However, p_3 has already forgotten its failure; furthermore, the check point does not check p_3 because it only includes processors reserved at the beginning of *make*. A similar flaw exists for creation instructions on entities with explicit processor specifications.

Clarification

This flaw can be resolved with another check point at the end of a creation call, as exemplified in Figure 6.25. In safe mode, the client p waits for the newly created processor q before releasing its request queue lock. For this purpose, it sets up a failure anchor: in case of a no failure, p simply releases the lock; in case of a failure, p releases the lock and then fails.

Clarification 6.3 (Failures). Before unlocking the request queue of a supplier q after a creation call in safe mode, the client p must query q for any failures and wait for the notification. □

Creation instruction (separate)

$$\frac{\begin{array}{l}(d,g,c) \overset{def}{=} type\_of(\Gamma, b)\\ g = \top \\ q \overset{def}{=} \sigma.new\_proc \\ o \overset{def}{=} \sigma.new\_obj(c) \\ \sigma' \overset{def}{=} \sigma.add\_proc(q).add\_obj(q,o) \\ r \overset{def}{=} \sigma'.ref(o) \\ a \text{ is fresh}\end{array}}{\Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1,\ldots,e_n); s_p, \sigma\rangle \rightarrow}$$

```
⟨p :: lock({q});
      write(b, r, false);
      b.f(e_1, ..., e_n);
      provided f.class_type.program.settings.safe_mode then
         issue(q, notify(a, false)); wait(a, q)
      else
         nop
      end;
      failure_anchor false
         final issue(q, unlock_rq); pop_obtained_locks
         normal nop
         rescue false nop
         retry nop
         failure fail
      end;
      s_p | q :: nop, σ'⟩
```

Figure 6.25: Safe mode clarification: creation

6.7 Related work

Section 6.1 classifies and discusses asynchronous exception mechanism found in the literature. This section zooms in on mechanism for SCOOP. Brooke and Paige [35] propose three options from which programmers can choose. The first two options are radical: the first option halts the entire system as a response to an asynchronous exception; the second option ignores any asynchronous exceptions. The third option marks failed objects permanently; any future client holds the failed object's handler accountable upon each feature call, i.e., each processor is an observer of each other processor, and an accountability persists until the end. The third option is limiting because it prevents cases where it would be valid to access the failed object once again. In particular, a different context might only assume that the failed object is consistent; it might not care about past failures. Hence, it would be valid to access the failed object again, provided that the failed object's consistency has been restored. Furthermore, the third option permits race conditions between the client and the supplier because the supplier is not permitted to finish its workload. On the plus side, polling at each feature call reduces the propagation delay.

In Arslan and Meyer's busy processor mechanism [13], a supplier becomes busy when it fails; only the client that made the faulty feature call can relock the request queue of the supplier, in which case it holds the supplier accountable. For this to work, the accountability persists beyond the locking context. This mechanism causes an exception to propagate in a context where the client cannot assume the promise of the corresponding feature call. Furthermore, it does not consider lock passing. If a client of a faulty supplier passes a lock to a second client, then the second client does not propagate the exception, even though it is operating in the locking context in which the exception occurred.

6.8 Discussion

This chapter introduced the accountability framework with concepts to describe asynchronous exception mechanisms and presented a classification of approaches along with a discussion of their strengths and weaknesses. It used them to derive a mechanism for asynchronous exceptions in SCOOP. With the proposed mechanism, the client only propagates an asynchronous exception when it synchronizes with the supplier. Upon synchronization, the client waits for the supplier to finish all previous asynchronous calls; hence it knows about any past failure, and thus no data race can occur. The failed supplier reports failures as long as its request queue remains locked. Once the client releases the lock, it can no longer rely on the postconditions of the called features because another processor can intervene.

Since a failure represents a failed postcondition, the supplier's failure becomes irrelevant, and thus the supplier can forget its failure. For applications where this is not suitable, the mechanism offers a safe mode where the client automatically synchronizes with a supplier before unlocking its request queue. This semantics ensures that no processor has to handle the supplier's failure in a context where the failure is irrelevant.

The mechanism can be extended with *exception resolution* [45]. A client can have more than one supplier, and each supplier can raise an asynchronous exception. In the proposed mechanism, the client handles the exception of the first supplier with whom it synchronizes. With exception resolution, the client can collect exceptions from multiple suppliers and handle them together. Examples with this technique include Rintala's C++ futures [186], SaGE [58], ProActive [48], the Guardian model [154], Arche [116], and Java ThreadGroups [80]. Further, the mechanism can be extended with a reset functionality for suppliers that failed asynchronously; this functionality can be invoked at the end of a rescue clause or during a retry instruction.

Another possible extension is *lock delegation*: a client p can delegate its obtained locks to another client q; after delegation, p asynchronously continues without the delegated locks. This mechanism is useful in case p wants q to continue its work on a supplier g without unlocking g's request queue and thus expiring g's accountability.

7 Data sharing

SCOOP_07 has no mechanism to share data between processors, and consequently, writing programs with shared data is clumsy. The shared data must be assigned to a new processor, handling client's access requests. For frequent and short read or write operations, this becomes problematic:

1. Each feature call to the data leads to a feature request in the request queue of the data processor, which then picks up the request and processes it on its call stack. This chain of actions creates a considerable overhead. For an asynchronous write operation, the overhead outweighs the benefit of asynchrony. For a synchronous read operation, the client not only waits for the data processor to process the request, it also gets delayed further by the overhead.

2. The data consumes operating system resources (threads, processes, locks, semaphores) that could otherwise be freed up.

On systems with shared memory, the clients can directly operate on the data, thus avoiding the overhead. This frees most of the operating system resources attached to the data processor. To address this need, this chapter applies the prototyping method on the development of a new mechanism for shared data: passive processors. It introduces passive processors informally, extends the formal specification with support for passive processors, and presents the testing results. Since performance cannot be evaluated with the executable formal specification alone, it uses a SCOOP implementation [67] with support for passive processors to experimentally demonstrate the performance benefits.

7.1 Informal description

Before accessing shared data, a processor must ensure its access is mutually exclusive; otherwise, data races can occur. For this purpose, shared data must be grouped, and each group must be protected through a lock. Since SCOOP processors offer this functionality already along with execution capabilities, one can use processors, stripped from their execution capabilities, to group and protect shared data.

Definition 7.1 (Passive processor). A *passive* processor q does not have any execution capabilities. Its request queue lock protects the access to its associated objects. A client p holding this lock uses feature calls to operate directly on q's associated objects. While operating on these objects, p assumes the identity of q. Processor q becomes passive when another processor sets it as passive. When q is not passive, it is *active*. Processor q becomes active again when another processor sets it as active. It can only become passive or active when its request queue is unlocked, i.e., when not being used by any other processor. □

When a processor p operates on the objects of a passive processor q, it assumes q's identity. For example, if p creates a literal object or another non-separate object, it creates this object on q and not on itself; otherwise, a non-separate entity on q would reference an object on p. Similarly, p copies or imports expanded objects to q, and it sets a once routine to not fresh on q.

Besides safe and fast data sharing, passive processors have further benefits:

- *Minimal user code changes.* The feature call primitive unifies sending messages to active processors and accessing shared data on passive processors, ensuring minimal code changes to set a processor passive or active. With respect to the type system, the same types can be used to type objects on passive and active processors. The existing type system rules ensure that no object on a passive processor can be seen as non-separate on a different processor, thus providing type soundness.

- *Minimal compiler and runtime changes.* To implement passive processors, much of the existing infrastructure can be reused. In particular, no new code for grouping objects and for locking request queues is required.

A pipeline system is a good representative for the class of programs targeted by the proposed mechanism: multiple stages share packages of data. The pipeline parallel design pattern [142] applies whenever a computation involves sending *packages* of data through a sequence of *stages* that operate on the packages. The pattern assigns each stage to a different thread, then the stages synchronize with each other to process the packages in the correct order. Using this pattern, each stage can be mapped for instance to a CPU core, a GPU core, an FPGA, or a cryptographic accelerator, depending on the stage's computational needs.

To implement a pipeline system in SCOOP, each stage and each package must be handled by its own processor to ensure that stages can access the packages in parallel. Each stage is numbered to indicate its position in the pipeline, and it receives this position upon creation:

```
class STAGE create make feature
  position: INTEGER -- The stage's position in the pipeline.

  make (new_position: INTEGER)
      -- Create a stage at the given position.
    do
      position := new_position
    end

  process (package: separate PACKAGE)
      -- Process the package after the previous stage is done with it.
    require
      package.is_processed (position − 1)
    do
      do_work (package) -- Read from and write to the package.
      package.set_processed (position) -- Set the package processed.
    end
end
```

To process each package in the right order, the stages must synchronize with each other. For this purpose, each package has two features *is_processed* and *set_processed* to keep track of the stages that already processed the package. The synchronization requirement can then be expressed elegantly using SCOOP's preconditions. The precondition in *process* delays the execution until the package has been processed by the previous stage.

```
class PACKAGE create make feature
  number_of_stages: INTEGER -- The number of stages.
  record: RECORD -- The processing history.

  make (new_number_of_stages: INTEGER)
      -- Create a package in a pipeline with the given number of stages.
    do
      number_of_stages := new_number_of_stages
```

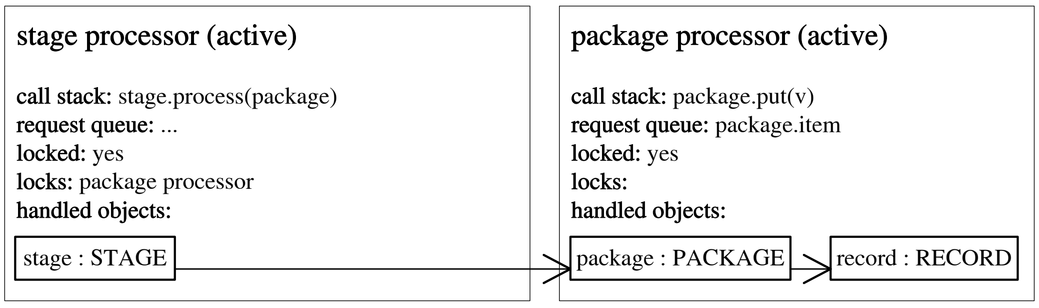
```
        create record.make (1, new_number_of_stages)
    end

  set_processed (stage_number: INTEGER)
        -- Set the package processed by the given stage.
    do
        record.put (True, stage_number)
    end

  is_processed (stage_number: INTEGER): BOOLEAN
        -- Is the package processed by the given stage?
    do
        if stage_number >= 1 then
            Result := record.item (stage_number)
        else
            Result := True
        end
    end
end
```

To avoid the messaging overhead of active processors, packages can be handled by passive processors. To achieve this, it suffices to set a package passive after its construction. The following code creates a package on a new passive processor and asks the stages to process the package.

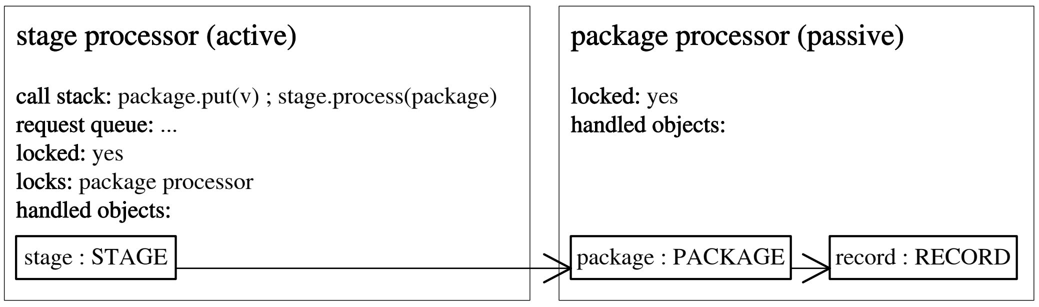
```
create package.make (data, number_of_stages); set_passive (package)
stage_1.process (package); ... ; stage_n.process (package)
```

No other code changes are necessary. The existing feature calls to the packages automatically assume the data access semantics. Furthermore, a stage can still use **separate** *PACKAGE* as the type of a package because the stage is still handled by a different processor than the package. Similarly, a package can still use the type *RECORD* for its record because the package still has the same handler as the record.

Figure 7.1 illustrates the effect of the call to *set_passive*. In Figure 7.1a, active package processors have a call stack, a request queue, and a stack of locks. The stage processors send asynchronous (see *put*) and synchronous (see *item*) feature requests. In Figure 7.1b, passive package processors do not have any execution capabilities. Therefore, the stage processors operate directly and synchronously on the packages, thus making a better use of their own processing capabilities rather than relaying all operations to the package processors.



(a) The package processors are active.



(b) The package processors are passive.

Figure 7.1: A stage processor processes a package. The stage object, handled by the left-hand side processor, has a separate reference to the package object, handled by the right-hand side processor. The package object references a non-separate record object to remember the processing history.

7.2 Formal specification

To operate on an object of a passive processor, a client assumes the identity of the passive processor. This section describes the changes to the formal specification.

7.2.1 Regions

To record passiveness, *REGIONS* has a query *passive* that returns whether a processor p is passive, as shown in Figure 7.2. It also has a command *set_passive* to set p passive ($ip = true$) or active ($ip = false$). When p operates on an object of a passive processor q, p assumes q's identity. For this purpose, *REGIONS* has a query *executes_for* and a command *set_executes_for*. When p operates on q's objects, the query *executes_for*(p) returns q. When p operates on its own objects, *executes_for*(p) returns p. Processor p is active upon creation and assumes its own identity, as shown in *add_proc*. The state facade mirrors these changes.

```
$passive\colon REGIONS \to PROC \nrightarrow BOOLEAN$
    $k.passive(p)$ **require**
        $k.procs.has(p)$
$executes\_for\colon REGIONS \to PROC \nrightarrow PROC$
    $k.executes\_for(p)$ **require**
        $k.procs.has(p)$
$set\_passive\colon REGIONS \to PROC \nrightarrow BOOLEAN \to REGIONS$
    $k.set\_passive(p, ip)$ **require**
        $k.procs.has(p)$
        $ip \Rightarrow k.executes\_for(p) = p$
        $\neg ip \Rightarrow \neg\exists q \in k.procs\colon (q \neq p \land k.executes\_for(q) = p)$
    **axioms**
        $k.set\_passive(p, ip).passive(p) = ip$
$set\_executes\_for\colon REGIONS \to PROC \nrightarrow PROC \nrightarrow REGIONS$
    $k.set\_executes\_for(p, q)$ **require**
        $k.procs.has(p)$
        $k.procs.has(q)$
        $p \neq q \Rightarrow \neg k.passive(p) \land k.passive(q)$
    **axioms**
        $k.set\_executes\_for(p, q).executes\_for(p) = q$
$add\_proc\colon REGIONS \to PROC \nrightarrow REGIONS$
    $k.add\_proc(p)$ **require**
        $\neg k.procs.has(p)$
    **axioms**
        $k.add\_proc(p).procs.has(p)$
        $k.add\_proc(p).last\_added\_proc = p$
        $k.add\_proc(p).handled\_objs(p).empty$
        $\neg k.add\_proc(p).rq\_locked(p)$
        $k.add\_proc(p).cs\_locked(p)$
        $k.add\_proc(p).obtained\_rq\_locks(p).empty$
        $k.add\_proc(p).obtained\_cs\_lock(p) = p$
        $k.add\_proc(p).retrieved\_rq\_locks(p).empty$
        $k.add\_proc(p).retrieved\_cs\_locks(p).empty$
        $\neg k.add\_proc(p).passed(p)$
        $\neg k.add\_proc(p).has\_failed(p)$
        $\neg k.add\_proc(p).passive(p)$
        $k.add\_proc(p).executes\_for(p) = p$
```

Figure 7.2: *REGIONS*: passive processor support

7.2.2 Writing and reading mechanism

When a processor p copies an expanded object while operating on behalf of a passive processor, p must create the copy on the passive processor. Figure 7.3 reflects this change. Processor p creates the copy on $executes\_for(p)$: if p operates on an object of q, the copy is handled by q; otherwise it is handled by p.

Write

$$\frac{\begin{array}{l}(\sigma', v') \overset{def}{=} \begin{cases}\text{if} \\ \quad ce \wedge \\ \quad v \in REF \wedge v \neq void \wedge \sigma.ref\_obj(v).class\_type.is\_exp \wedge \\ \quad \sigma.handler(v) = \sigma.executes\_for(p) \\ \text{then} \\ \quad (\sigma^*, \sigma^*.last\_added\_obj) \\ \quad\quad \textbf{where} \\ \quad\quad\quad \sigma^* \overset{def}{=} \sigma.add\_obj(\sigma.executes\_for(p), \sigma.ref\_obj(v).copy) \\ \text{otherwise} \\ \quad (\sigma, v)\end{cases} \\ \sigma'' \overset{def}{=} \sigma'.set\_val(p, b.name, v')\end{array}}{\Gamma \vdash \langle p :: \texttt{write}(b, v, ce); s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma''\rangle}$$

Figure 7.3: Writing and reading mechanism: passive processor support

7.2.3 Literal expressions

When p creates a literal object, p also creates the object on the processor returned by $executes\_for(p)$. Figure 7.4 shows the change.

Literal expression

$$\frac{\begin{array}{l}e \in LITERAL \\ \sigma' \overset{def}{=} \begin{cases}\sigma & \text{if } e = \textbf{Void} \\ \sigma.add\_obj(\sigma.executes\_for(p), obj(e)) & \text{otherwise}\end{cases} \\ r \overset{def}{=} \begin{cases}void & \text{if } e = \textbf{Void} \\ \sigma'.last\_added\_obj & \text{otherwise}\end{cases}\end{array}}{\Gamma \vdash \langle p :: \texttt{eval}(a, e); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a, r); s_p, \sigma'\rangle}$$

Figure 7.4: Literal expressions: passive processor support

7.2.4 *Feature calls*

To set a processor q passive, another processor p executes $\mathit{set\_passive}(e)$ with an expression e referencing an object on q. Processor p evaluates e and performs a runtime call, as described in Figure 7.5. The runtime call only succeeds once q's request queue is unlocked. Once q is passive, p operates directly on q's objects. When p performs a call on one of q's objects, it does not send a feature request to q; instead, p processes the feature request itself, as shown in Figure 7.6 for query calls. Processor p first determines the handler q of the target and the processor g to apply the called feature, i.e., p if q is passive and q otherwise. Processor p then imports and copies arguments of expanded type to q.

Call runtime (set passive)

$$\frac{\neg\sigma.\mathit{rq\_locked}(\sigma.\mathit{handler}(r_1)) \qquad \sigma' \overset{def}{=} \sigma.\mathit{set\_passive}(\sigma.\mathit{handler}(r_1), \mathit{true})}{\Gamma \vdash \langle p :: \mathtt{call}(r_0, \mathit{set\_passive}, (r_1)); s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

Call runtime (set active)

$$\frac{\neg\sigma.\mathit{rq\_locked}(\sigma.\mathit{handler}(r_1)) \qquad \sigma' \overset{def}{=} \sigma.\mathit{set\_passive}(\sigma.\mathit{handler}(r_1), \mathit{false})}{\Gamma \vdash \langle p :: \mathtt{call}(r_0, \mathit{set\_active}, (r_1)); s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma'\rangle}$$

Figure 7.5: Runtime calls: passive processor support

7.2.5 *Feature applications*

To apply a feature f on behalf of a passive processor g, p assumes g's identity, as shown in Figure 7.7, Figure 7.8, and Figure 7.9. Processor p is not required to be the handler of the target, as it is the case for calls to active processors; instead, p must hold a request queue lock on g (see premise). To restore its identity later on, p saves the identity (see w). It then proceeds on behalf of q using its own execution capabilities, e.g., its stacks of variable environments and locks. If f is a fresh once routine, p sets f to not fresh on q. If f is a not fresh once routine, p queries f's status on q and proceeds accordingly. To return a result of expanded type to a different processor, p imports the result to $\mathit{executes\_for}(q)$, as described in Figure 7.10. If the client q is operating on a passive processor, p imports the result to the passive processor; otherwise it imports the result to q. Finally, p restores its previous identity. For this purpose, the `return` operation receives a new argument w with the previous identity, as shown in Figure 7.10 for queries.

Call feature (query)

$$\frac{\begin{array}{l}
q \overset{def}{=} \sigma.handler(r_0) \\
g \overset{def}{=} \begin{cases} p & \text{if } \sigma.passive(q) \\ q & \text{otherwise} \end{cases} \\
\bar{l} \overset{def}{=} \begin{cases} (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) & \text{if } g \neq p \\ (\{\}, \{\}) & \text{otherwise} \end{cases} \\
\sigma'_0 \overset{def}{=} \sigma \\
\forall i \in \{1, \ldots, n\}\colon (\sigma'_i, r'_i) \overset{def}{=} \\
\quad \begin{cases}
\text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) \neq q \\
\quad (\sigma^*, \sigma^*.last\_imported\_obj) \\
\quad\quad \textbf{where} \\
\quad\quad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.import(q, r_i) \\
\text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) = q \\
\quad (\sigma^*, \sigma^*.last\_added\_obj) \\
\quad\quad \textbf{where} \\
\quad\quad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.add\_obj(q, \sigma'_{i-1}.ref\_obj(r_i).copy) \\
\text{otherwise} \\
\quad (\sigma'_{i-1}, r_i)
\end{cases}
\end{array}}{\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{call}(a, r_0, f, (r_1, \ldots, r_n)); s_p, \sigma \rangle \rightarrow \\
\quad \langle p :: \texttt{issue}(g, \texttt{apply}(a, r_0, f, (r'_1, \ldots, r'_n), p, \bar{l})); \texttt{wait}(a, g); s_p, \sigma'_n \rangle
\end{array}}$$

Figure 7.6: Query calls: passive processor support

Apply feature (non-once routine or fresh once routine)

$$
\begin{array}{l}
w \overset{def}{=} \sigma.executes\_for(p) \\
\sigma' \overset{def}{=} \sigma.set\_executes\_for(p, \sigma.handler(r_0)) \\
f \in ROUTINE \wedge (f.is\_once \Rightarrow \sigma.fresh(\sigma'.executes\_for(p), f.id)) \\
\neg\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \sigma'.handler(r_0) = p \\
\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \neg\sigma'.passed(p) \wedge \sigma'.rq\_locks(p).has(\sigma'.handler(r_0)) \\
\sigma'' \overset{def}{=} \begin{cases} \text{if } f \in FUNCTION \wedge f.is\_once \\ \quad \sigma'.set\_once\_func\_not\_fresh(\sigma'.executes\_for(p), f, false, false, void) \\ \text{if } f \in PROCEDURE \wedge f.is\_once \\ \quad \sigma'.set\_once\_proc\_not\_fresh(\sigma'.executes\_for(p), f, false, false) \\ \text{otherwise} \quad \sigma' \end{cases} \\
\sigma''' \overset{def}{=} \sigma''.pass\_locks(q, p, (\bar{l}_r, \bar{l}_c)).push\_env\_with\_feature(p, f, r_0, (r_1, \ldots, r_n)) \\
\overline{g}_{required\_rq\_and\_cs\_locks} \overset{def}{=} \{p\} \cup \\
\quad \{x \in PROC \mid \exists i \in \{1, \ldots, n\}, g, c : \Gamma \vdash f.formals(i) : (!, g, c) \wedge \\
\quad \sigma'''.ref\_obj(r_i).class\_type.is\_ref \wedge x = \sigma'''.handler(r_i)\} \\
\overline{g}_{required\_cs\_locks} \overset{def}{=} \\
\quad \{x \in \overline{g}_{required\_rq\_and\_cs\_locks} \mid x = p \vee \\
\quad (x \neq p \wedge \sigma'''.passed(x) \wedge \neg\sigma'''.passed(p) \wedge \sigma'''.cs\_locks(p).has(x)\} \\
\overline{g}_{required\_rq\_locks} \overset{def}{=} \overline{g}_{required\_rq\_and\_cs\_locks} \setminus \overline{g}_{required\_cs\_locks} \\
\{g_1, \ldots, g_m\} \overset{def}{=} \overline{g}_{missing\_rq\_locks} \overset{def}{=} \{x \in \overline{g}_{required\_rq\_locks} \mid \neg\sigma'''.rq\_locks(p).has(x)\} \\
a' \text{ is fresh} \\
\hline
\Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow \\
\quad \langle p :: \texttt{check\_pre\_and\_lock}(a, f, q, (\bar{l}_r, \bar{l}_c), \overline{g}_{missing\_rq\_locks}, w); \\
\qquad \texttt{execute\_body}(f, \overline{g}_{missing\_rq\_locks}); \texttt{check\_post\_and\_inv}(f); \\
\qquad \texttt{failure\_anchor}\ false \\
\qquad\quad \texttt{final} \\
\qquad\qquad \texttt{issue}(g_1, \texttt{unlock\_rq}); \ldots; \texttt{issue}(g_m, \texttt{unlock\_rq}); \\
\qquad\qquad \texttt{pop\_obtained\_locks} \\
\qquad\quad \texttt{normal} \\
\qquad\qquad \texttt{provided}\ f \in FUNCTION\ \texttt{then} \\
\qquad\qquad\quad \texttt{read}(result, a'); \texttt{return}(a, f, a'.data, q, (\bar{l}_r, \bar{l}_c), w, false, true, false) \\
\qquad\qquad\ \texttt{else}\ \texttt{return}(a, f, q, (\bar{l}_r, \bar{l}_c), w, false, true, false)\ \texttt{end} \\
\qquad\quad \texttt{rescue}\ true\ \texttt{execute\_rescue\_clause}(f) \\
\qquad\quad \texttt{retry}\ \texttt{execute\_body}(f, \overline{g}_{missing\_rq\_locks}); \texttt{check\_post\_and\_inv}(f) \\
\qquad\quad \texttt{failure}\ \texttt{return}(a, f, q, (\bar{l}_r, \bar{l}_c), w, true, true, false) \\
\qquad \texttt{end}; s_p, \sigma'''\rangle
\end{array}
$$

Figure 7.7: Non-once or fresh once routine application: passive processor support

Apply feature (not fresh once routine)

$$\frac{\begin{array}{l} w \overset{def}{=} \sigma.executes\_for(p) \\ \sigma' \overset{def}{=} \sigma.set\_executes\_for(p, \sigma.handler(r_0)) \\ f \in ROUTINE \wedge f.is\_once \wedge \neg\sigma'.fresh(\sigma'.executes\_for(p), f.id) \wedge \\ \quad (\sigma'.stable(\sigma'.executes\_for(p), f.id) \vee \\ \quad \sigma'.stabilizer(\sigma'.executes\_for(p), f.id) = \sigma'.executes\_for(p)) \\ \neg\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \sigma'.handler(r_0) = p \\ \sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \neg\sigma'.passed(p) \wedge \sigma'.rq\_locks(p).has(\sigma'.handler(r_0)) \\ \sigma'' \overset{def}{=} \sigma'.pass\_locks(q, p, (\bar{l}_r, \bar{l}_c)).push\_env\_with\_feature(p, f, r_0, (r_1, \dots, r_n)) \end{array}}{\Gamma \vdash \langle p :: \mathtt{apply}(a, r_0, f, (r_1, \dots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow}$$

```
    ⟨p :: provided $\sigma''.has\_failed(\sigma''.executes\_for(p), f)$ then
            fail
        else
            nop
        end;
        failure_anchor $false$
            final nop
            normal
                provided $f \in FUNCTION$ then
                    return($a, f, \sigma''.once\_result(\sigma''.executes\_for(p), f.id),$
                        $q, (\bar{l}_r, \bar{l}_c), w, false, false, false$)
                else
                    return($a, f, q, (\bar{l}_r, \bar{l}_c), w, false, false, false$)
                end
            rescue $false$ nop
            retry nop
            failure return($a, f, q, (\bar{l}_r, \bar{l}_c), w, true, false, false$)
        end;
        $s_p, \sigma''$⟩
```

Figure 7.8: Not fresh once routine application: passive processor support

Apply feature (attribute)

$$\dfrac{\begin{array}{l} w \overset{def}{=} \sigma.executes\_for(p) \\ \sigma' \overset{def}{=} \sigma.set\_executes\_for(p, \sigma.handler(r_0)) \\ f \in ATTRIBUTE \\ \neg\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \sigma'.handler(r_0) = p \\ \sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \neg\sigma'.passed(p) \wedge \sigma'.rq\_locks(p).has(\sigma'.handler(r_0)) \\ \sigma'' \overset{def}{=} \sigma'.pass\_locks(q, p, (\bar{l}_r, \bar{l}_c)).push\_env\_with\_feature(p, f, r_0, ()) \\ a' \text{ is fresh} \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a', f); \\ \qquad \texttt{wait}(a', p); \\ \qquad \texttt{return}(a, f, a'.data, q, (\bar{l}_r, \bar{l}_c), w, false, false, false); \\ \qquad s_p, \sigma''\rangle \end{array}}$$

Figure 7.9: Attribute application: passive processor support

$$
\begin{aligned}
&\mathit{RETURN} \triangleq \ldots \mid \\
&\quad \texttt{return}\ \mathit{TUPLE}\langle \mathit{CHANNEL},\ \mathit{FEATURE},\ \mathit{REF},\ \mathit{PROC}, \\
&\qquad \mathit{TUPLE}\langle \mathit{SET}\langle \mathit{PROC}\rangle,\ \mathit{SET}\langle \mathit{PROC}\rangle\rangle,\ \mathit{PROC}, \\
&\qquad \mathit{BOOLEAN},\ \mathit{BOOLEAN},\ \mathit{BOOLEAN}\rangle\ ;
\end{aligned}
$$

Return (successful query)

$$
\frac{
\begin{array}{l}
(\sigma', r_r') \overset{def}{=} \begin{cases}
\text{if} \\
\quad r_r \neq void \wedge \sigma.ref\_obj(r_r).class\_type.is\_exp \wedge \\
\quad \sigma.handler(r_r) \neq \sigma.executes\_for(q) \\
\text{then} \\
\quad (\sigma^*, \sigma^*.last\_imported\_obj) \\
\qquad \textbf{where} \\
\qquad\quad \sigma^* \overset{def}{=} \sigma.import(\sigma.executes\_for(q), r_r) \\
\text{otherwise} \\
\quad (\sigma, r_r)
\end{cases} \\
\sigma'' \overset{def}{=} \begin{cases}
\text{if } f.is\_once \wedge snf \\
\quad \sigma'.set\_once\_func\_not\_fresh(\sigma.executes\_for(p), f, true, false, r_r) \\
\text{if } f.is\_once \wedge sf \\
\quad \sigma'.set\_once\_rout\_fresh(\sigma.executes\_for(p), f) \\
\text{otherwise} \\
\quad \sigma'
\end{cases} \\
\sigma''' \overset{def}{=} \sigma''.pop\_env(p).revoke\_locks(q, p).set\_executes\_for(p, w)
\end{array}
}{
\Gamma \vdash \langle p :: \texttt{return}(a, f, r_r, q, (\bar{l}_r, \bar{l}_c), w, false, snf, sf); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a, r_r'); s_p, \sigma'''\rangle
}
$$

Figure 7.10: Query return: passive processor support

7.2.6 *Creation instructions*

When p creates a non-separate object while operating on behalf of a passive processor, p creates the object on the processor returned by $executes\_for(p)$, as shown in Figure 7.11.

Creation instruction (non-separate)

$$\frac{\begin{array}{l}(d,g,c) \overset{def}{=} type\_of(\Gamma,b)\\ g = \bullet\\ o \overset{def}{=} \sigma.new\_obj(c)\\ \sigma' \overset{def}{=} \sigma.add\_obj(\sigma.executes\_for(p),o)\\ r \overset{def}{=} \sigma'.ref(o)\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1,\ldots,e_n);\ s_p,\sigma\rangle \rightarrow\\ \quad \langle p :: \texttt{write}(b,r,false);\\ \qquad b.f(e_1,\ldots,e_n);\\ \qquad s_p,\sigma'\rangle\end{array}}$$

Figure 7.11: Non-separate creation instruction: passive processor support

7.3 Testing

The test suite from Chapter 5 also includes tests that combine passive processors with other SCOOP aspects. These tests revealed one flaw. A supplier p calls the import feature on an object of expanded type before returning it to a different processor. If the client q is operating on a passive processor, p imports the result to the passive processor; otherwise it imports the result to q.

7.3.1 *Flaw*

If a processor p returns an expanded object from a passive processor g to itself or another passive processor, it does not import the result (see Figure 7.10). This negligence is problematic, as the following code shows:

```
class A create make feature
  make
    local
      b: separate B
    do
```

```
      create b.make;
      set_passive (b)
      f (b)
    end

  f (b: separate B)
    local
      i: INTEGER
    do
      i := b.g
    end
end

class B create make feature
  make
    do end

  g: INTEGER
    do
      Result := 0
    end
end
```

Using Maude's **rew** command, a new root processor p_1 creates a new object on a new passive processors p_3. It starts executing the feature f and locks p_3's request queue to execute the query $b.g$ on behalf of p_3 (see Figure 7.7). It saves its identity, i.e., $w = p_1$ and assumes p_3's identity, i.e., $executes\_for(p_1) = p_3$. It then creates a new expanded object with reference r on p_3, i.e., $handler(r) = p_3$, and assigns r to the result entity. When p_1 returns (see Figure 7.10), it does not import r because $handler(r) = p_3 = executes\_for(p_1)$. Hence p_1 receives an expanded object that it supposed to be handled by p_1, but is actually handled by p_3.

7.3.2 Clarification

If p returns an expanded object from a passive processor to itself or another passive processor, it must import the object to that processor. Figure 7.12 shows the clarified `return` operation. Processor p determines the processor g to which p imports the expanded object. If p finishes a call to a passive processor, then p imports to w, i.e., itself or another passive processor; otherwise, p behaves as before.

Clarification 7.1 (Passive processors). If p returns an expanded object with reference r from a passive processor to itself or another passive processor, it must import r to that processor. □

Return (successful query)

$$g \overset{def}{=} \begin{cases} w & \text{if } p = q \\ \sigma.executes\_for(q) & \text{otherwise} \end{cases}$$

$$(\sigma', r_r') \overset{def}{=} \begin{cases} \text{if } r_r \neq void \wedge \sigma.ref\_obj(r_r).class\_type.is\_exp \wedge \sigma.handler(r_r) \neq g \\ \quad (\sigma^*, \sigma^*.last\_imported\_obj) \\ \quad\quad \textbf{where} \\ \quad\quad\quad \sigma^* \overset{def}{=} \sigma.import(g, r_r) \\ \text{otherwise} \\ \quad (\sigma, r_r) \end{cases}$$

$$\sigma'' \overset{def}{=} \begin{cases} \text{if } f.is\_once \wedge snf \\ \quad \sigma'.set\_once\_func\_not\_fresh(\sigma.executes\_for(p), f, true, false, r_r) \\ \text{if } f.is\_once \wedge sf \\ \quad \sigma'.set\_once\_rout\_fresh(\sigma.executes\_for(p), f) \\ \text{otherwise} \\ \quad \sigma' \end{cases}$$

$$\sigma''' \overset{def}{=} \sigma''.pop\_env(p).revoke\_locks(q, p).set\_executes\_for(p, w)$$

$$\frac{}{\Gamma \vdash \langle p :: \texttt{return}(a, f, r_r, q, (\bar{l}_r, \bar{l}_c), w, false, snf, sf); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a, r_r'); s_p, \sigma'''\rangle}$$

Figure 7.12: Passive processor return clarification

7.4 Performance evaluation

Improved performance is one of the prime goals of the passive processor mechanism. Since performance cannot be evaluated with the executable formal specification alone, this section uses a SCOOP implementation [67] with support for passive processors to experimentally evaluate the performance of the mechanism. Since the pipeline system is a good representative for the class of targeted programs, this section compares the performance of the pipeline system when implemented using passive processors, active processors, and low-level synchronization primitives; the latter two are the closest competing approaches.

Table 7.1: Average execution times (in seconds) of various low-pass filter systems with various signal lengths

configuration	2048	4096	8192	16384	32768	65536	131072	262144	524288
sequential, SCOOP	1.00	1.66	3.22	6.35	12.71	26.19	55.05	120.37	272.38
sequential, thread	0.62	1.09	2.19	4.66	9.57	20.23	41.45	93.40	213.59
1 pipeline, active	337.55	682.29	1456.67	2875.23	-	-	-	-	-
1 pipeline, passive	1.64	2.72	5.02	10.99	24.68	55.29	118.30	247.29	533.83
1 pipeline, thread	0.31	0.58	1.16	2.44	5.26	11.11	23.59	53.24	122.00
2 pipelines, passive	1.19	1.77	3.05	6.35	14.19	29.82	60.24	124.99	263.96
3 pipelines, passive	1.07	1.47	2.34	4.71	9.87	20.65	41.78	85.72	185.19
5 pipelines, active	231.25	496.68	1048.95	2192.53	-	-	-	-	-
5 pipelines, passive	0.87	1.23	1.86	3.27	6.35	13.50	27.17	55.95	117.12
5 pipelines, thread	0.16	0.23	0.40	0.74	1.53	3.05	6.30	13.76	31.82
10 pipelines, active	334.93	726.83	1549.01	3322.70	-	-	-	-	-
10 pipelines, passive	0.84	1.08	1.38	2.36	4.13	8.28	16.89	35.13	76.64
10 pipelines, thread	0.16	0.22	0.33	0.59	1.08	2.09	4.26	9.21	20.93

7.4.1 Comparison to active processors

A low-pass filter pipeline is especially suited because it exhibits frequent and short read and write operations on the packages, each of which represents a signal to be filtered. The pipeline has three stages: the first performs a decimation-in-time radix-2 fast Fourier transformation [117]; the second applies a low-pass filter in Fourier space; and the third inverses the Fourier transformation. The system supports any number of pipelines operating in parallel, and it splits the signals evenly.

Table 7.1 shows the average execution times of various low-pass filter systems processing signals of various lengths. The experiments have been conducted on a 4 × Intel Xeon E7-4830 2.13 GHz server (32 physical cores), 256 GB of RAM running Windows Server 2012 Datacenter (64 Bit) in a Kernel-based Virtual Machine (KVM) on Red Hat 4.4.7-3 (64 Bit). A modified version of EVE 13.11 [67] compiled the programs in finalized mode with an inline depth of 100. Every data point reflects the average execution time over ten runs processing 100 signals each. Using ten pipelines, it took nearly ten hours to compute the average for active signals of length 16384; thus we refrained from computing data points for bigger lengths.

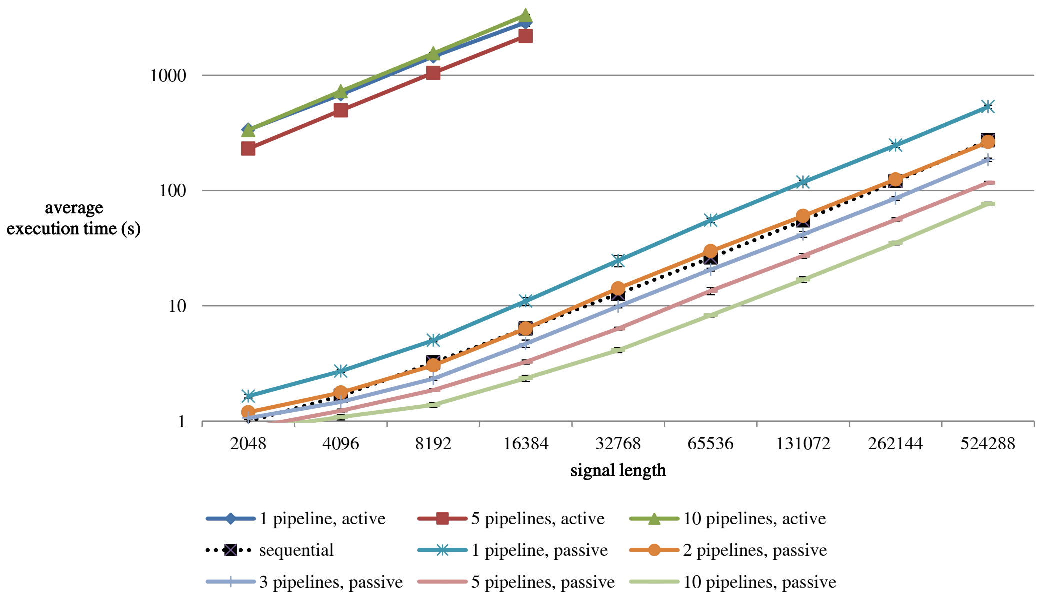


Figure 7.13: Speedup of passive processors over active processors

Figure 7.13 visualizes the data. The upper three curves belong to the active signal processors. The lower curves result from the passive processors and a sequential execution. As the graph indicates, the passive processors are more than two orders of magnitude faster than the active ones. In addition, with increasing number of pipelines, the passive processors become faster than the sequential program. In fact, two pipelines are enough to have an equivalent performance. The overhead of the mechanism is thus small enough to benefit from an increase in parallelism. In contrast, active processors deliver their peak performance with around five pipelines but never get faster than the sequential programs.

7.4.2 Comparison to low-level synchronization primitives

Figure 7.14 and Table 7.1 compare pipelines with passive processors to pipelines based on low-level synchronization primitives. In the measured range, the passive processors are between 3.7 to 5.4 times slower. As the signal length increases, the slowdown tends to becomes smaller. With more pipelines, the slowdown also tends to decrease at signal lengths above 8192. The two curves for sequential executions show that a slowdown can also be observed for non-concurrent programs.

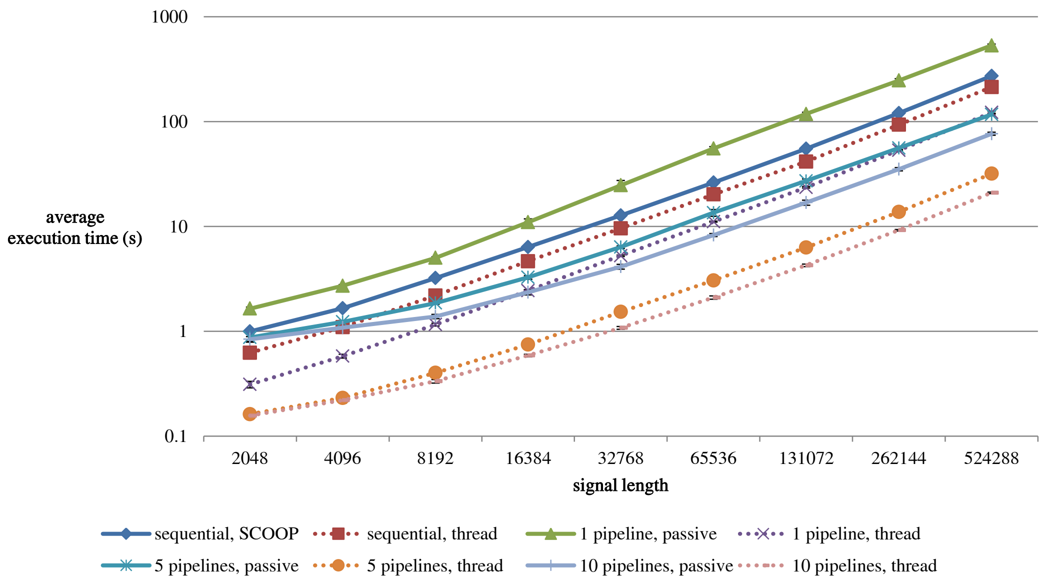


Figure 7.14: Slowdown of passive processors over EiffelThread

The slowdown is the consequence of SCOOP's programming abstractions. Compare the following thread-based stage implementation to the SCOOP one from Section 7.1. Besides the addition of boilerplate (inherit clause, redefinition of *execute*), this code exhibits some more momentous differences. First, the thread-based stage class implements a work queue: it has an attribute to hold the packages and a loop in *execute* to go over them. In SCOOP, request queues provide this functionality. Second, each thread-based package has a mutex and a condition variable for synchronization. To process a package, stage i first locks the mutex and then uses the condition variable to wait until stage $i - 1$ has processed the package. Once stage $i - 1$ is done, it uses the condition variable to signal all waiting stages. Only stage i leaves the loop. In SCOOP, wait conditions provide this kind of synchronization off-the-shelf. We expect the cost of wait conditions and other aspects to drop further as the compiler and the runtime mature.

```
class STAGE inherit THREAD create make feature
  position: INTEGER -- The stage's position in the pipeline.
  packages: ARRAY[PACKAGE] -- The packages to be processed.

  make (new_position: INTEGER; new_packages: ARRAY[PACKAGE])
      -- Create a stage at the given position to operate on the packages.
    do
      position := new_position
      packages := new_packages
    end

  execute
      -- Process each package after the previous stage is done with it.
    do
      across packages as package loop
        package.mutex.lock -- Lock the package.
        -- Sleep until previous stage is done; release the lock meanwhile.
        from until package.is_processed (position − 1) loop
          package.condition_variable.wait (package.mutex)
        end
        process (package) -- Process the package.
        package.condition_variable.broadcast -- Wake up next stage.
        package.mutex.unlock -- Unlock the package.
      end
    end

  process (package: PACKAGE)
      -- Process the package.
    do
      do_work (package) -- Read from and write to the package
      package.set_processed (position) -- Set the package processed.
    end
end
```

7.5 Other applications

A variety of other applications could also profit from passive processors. Object structures can be distributed over passive processors. Multiple clients can thus operate on dynamically changing but distinct parts of these structures while exchanging coordination messages. For example, in parallel graph algorithms, the vertices can be distributed over passive processors. In producer-consumer programs, intermediate buffers can be passive. Normally, about half of the operations in producer-consumer programs are synchronous read accesses. Without the messaging overhead, the consumer can execute these operations much faster than the buffer. Passive processors can also be useful to handle objects whose only purpose it is to be lockable, e.g., forks of dining philosophers, or to encapsulate a shared state, e.g., a robot's state in a controller [184].

7.6 Related work

Several languages and libraries combine shared data with message passing. In Ada [115], *tasks* execute concurrently and communicate during a *rendezvous*: upon joining a rendezvous, the client waits for a message from the supplier, and the supplier synchronously sends a message to the client. The client joins a rendezvous by calling a supplier's *entry*. The supplier joins by calling an *accept* statement on that entry. To share data, tasks access *protected objects* that encapsulate data and provide exclusive access thereon through guarded functions, procedures, and entries. Since functions may only read data, multiple function calls may be active simultaneously. In contrast, passive processors do not support multiple readers. However, unlike protected objects, passive processors do not require new data access primitives. Furthermore, passive processors can become active at runtime.

Erlang [66] is a functional programming language whose concurrency support is based on the actor model [89]. Processes exchange messages and share data using an *ets table*, providing atomic and isolated access to table entries. A process can also use the *Mnesia* database management system to group a series of table operations into an atomic transaction. While passive processors do not provide support for transactions, they are not restricted to tables.

Schill et al. [190] developed a library offering indexed arrays that can be accessed concurrently by multiple SCOOP processors. To prevent data races on an array, each processor must reserve a *slice* of the array. Slices support fine-grained sharing as well as multiple readers using *views*, but they are restricted to indexed containers. For instance, distributed graphs cannot be easily expressed.

A group of MPI [146] processes can share data using the *remote memory ac-*

cess mechanism and its *passive target* communication. The processes collectively create a *window* of shared memory. In a *static* window, each process dedicates a segment of its local memory and maps it into the window during creation. In a window with *shared segments*, a process accesses data using local memory operations, i.e., it maps the window into its local address space. With *non-shared segments*, a process accesses data using remote memory access operations, where data are identified by a process *rank*, i.e., its identifier, and an offset. In a *dynamic* window, processes attach and detach segments dynamically after creating the window. Processes access a window during an *epoch*, which begins with a collective synchronization call, continues with communication calls, and ends with another synchronization call. Synchronization includes fencing and locking. Locks can be partial or full, and they can be shared or exclusive. Passive processors neither offer fences nor shared locks; they do, however, offer automatic conditional synchronization based on preconditions. MPI can also be combined with OpenMP [172]. Just like MPI alone, this combination does not provide unified concepts. Instead, it provides distinct primitives to access shared data and to send messages.

Uniformity distinguishes passive processors also from further approaches. For example, Wong and Rendell [218] extend the Danui distributed shared memory system with message passing, thus supporting both communication styles. Gustedt [87] proposes an interface to transfer data and access rights thereon in a message-passing system. Kranz et al. [124] describe how the Alewife computer architecture and its software stack support shared data and message passing.

Several studies agree that performance gains can be realized if the setup of a program with both message passing and shared data fits the underlying architecture. For instance, Bull et al. [41], Cappello and Etiemble [46], as well as Rabenseifner et al. [182] focus on benchmarks for MPI+OpenMP. Wei and Yilmaz [214] study an adaptive integral method to analyze electromagnetic scattering from perfectly conducting surfaces. Chorley and Walker [52] study a molecular dynamics simulation. Dunn and Meyer [61] study a QR factorization algorithm that can be adjusted to apply only message passing, only shared data, or both.

A number of approaches focus on optimizing messaging on shared memory systems instead of combining message passing with shared data. Gruber and Boyer [84] use an ownership management system to avoid copying messages between actors while retaining memory isolation. When an actor sends a message, the system transfers ownership over the message to the receiver. From that point on, only the receiver can access the message, and the sender would get an exception if it tries to do so. Villard et al. [210], Negara et al. [162], and Bono et al. [27] employ static analysis techniques to determine when a message can be passed by reference rather than by value. Protopopov and Skjellum [181], Ávila et al. [15], Buntinas et al. [42, 43], Graham and Shipman [83], as well as Brightwell [34] present techniques to allocate and use shared memory for messages.

7.7 Discussion

A client calling a feature on an object of a passive processor does not send a feature request; instead, it operates directly on the object after obtaining the request queue lock of the passive processor. Passive processors extend SCOOP's message-passing foundation with support for safe data sharing, reducing the execution time by several orders of magnitude on data-intensive parallel programs. They are useful whenever multiple processors access shared data using frequent and short read or write operations, where the overhead outweighs the benefit of asynchrony.

Passive processors can be implemented with minimal effort because much of the existing infrastructure can be reused. The feature call primitive unifies sending messages to active processors and accessing shared data on passive processors. Therefore, no significant code change is necessary to set a processor passive or active. This smooth integration differentiates passive processors from other approaches. The concept can also be generalized and applied to other message-passing concurrency models. For instance, messages to passive actors [89] can be translated into direct, synchronous, and mutually exclusive accesses to the actor's data.

Passive processors can be developed further in several directions. With *shared read locks*, multiple clients can simultaneously operate on a passive processor. Shared read locks require features that are guaranteed to be read-only. Functions could serve as a first approximation since they are read-only by convention. Further, passive processors can be added to a future distributed version of SCOOP. Because frequent remote calls are expensive, implementing distributed passive processors requires an implicit copy mechanism to move the supplier's data into the client's memory.

8 Performance analysis

The executable formal specifications enables SCOOP to be tested; however, having a verified concurrency model does not ensure that programmers write efficient concurrent programs. As the complex interactions between processors make the analysis of the behavior of concurrent programs difficult, effective use of SCOOP requires tools to help programmers analyze and improve programs. Since SCOOP's execution semantics differs considerably from established models, reusing an established tool is not an option for SCOOP. The chapter presents performance metrics and a performance analyzer specifically designed for SCOOP. The metrics and the tool are evaluated on a number of example problems as well as on a larger case study on optimizing a robotics control software written in SCOOP.

8.1 Performance metrics

To define SCOOP-specific metrics, one must take a closer look at the interactions in the runtime system. Consider a share market application with investors, share issuers, and markets, where integer identifiers represent the issuers. The following listing shows the class that describes the investors. The market and the investors are handled by different processors.

```
class INVESTOR feature
  id: INTEGER -- The identifier.

  buy (market: separate MARKET; issuer_id: INTEGER)
      -- Buy a share of the issuer on the market.
    require
      market.can_buy (id, issuer_id)
    do
```

```
      market.buy (Current, issuer_id)
    end

  sell (market: separate MARKET; issuer_id: INTEGER)
      -- Sell a share of the issuer on the market.
    require
      market.can_sell (id, issuer_id)
    do
      market.sell (Current, issuer_id)
    end
end
```

An investor has features to buy and sell shares. To execute one of these features, the investor must wait for the request queue lock on the market and for the precondition to be satisfied. In the feature call to the market, the investor passes a reference to itself, which triggers lock passing. This enables the market to query the investor's identifier in a separate callback, but the lock passing operation forces the investor to wait until the market finished. Consider the following routine that makes an investor buy a share on a market. A log keeps a record of the transaction.

```
do_transaction (
  investor: separate INVESTOR;
  issuer_id: INTEGER;
  log: separate LOG
)
    -- Make the investor buy a share of the issuer on a market. Log the transaction.
  require
    not log.is_full
  do
    investor.buy (market, issuer_id)
    log.add_buy_entry (investor.id, issuer_id)
  end
```

Figure 8.1 illustrates a possible execution with one timeline for each processor. Looking at these timelines, one can deduce the time metrics in Definition 8.1.

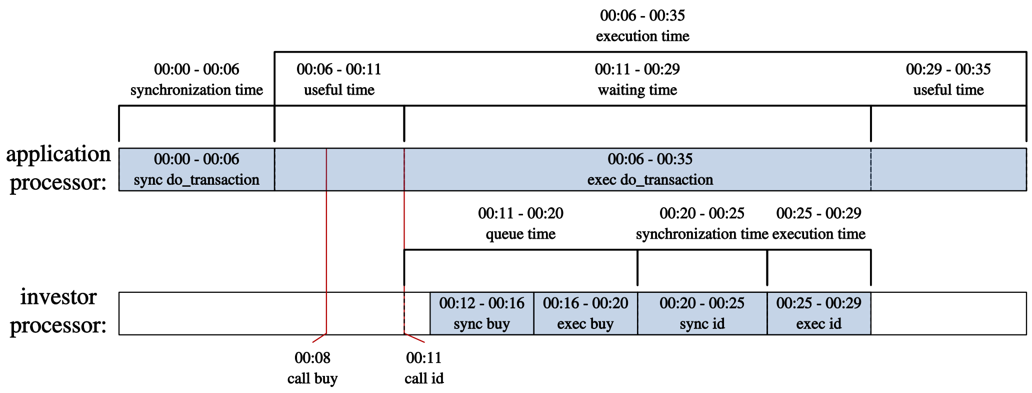


Figure 8.1: Time intervals for the share market application. The application first synchronizes (00:00 – 00:06) to get the request queue locks on its arguments' handlers and to ensure that the log is not full. It then executes (00:06 – 00:35) two feature calls and waits for the second one to complete. Each feature call leads to a synchronization and execution step in the investor.

Definition 8.1 (Time metrics). The following time metrics exist:

- *Lifetime* of a processor: the time from when the processor gets created until when it is no longer needed, at which point it will be reclaimed by the runtime system.
- *Queue time* of a feature request: the time from the feature call until the feature application begins. For non-separate feature calls, the queue time is zero.
- *Synchronization time* of a feature request: the time from when the feature application begins until when all the required request queues are locked and the precondition is satisfied.
- *Execution time* of a feature request: the time during which the feature actually gets executed. It is divided into the following parts:
  - *Waiting time*: the sum of
    - the queue time, the synchronization time, and the execution time of each feature request that results from a synchronous separate feature call, and
    - the synchronization time of each feature request that results from a non-separate feature call.

  - *Useful time*: the remaining part of the execution time in which the executing processor is not waiting.

- *Measuring time* of a processor: the time during which the processor performs measuring related actions.
- *Idle time* of a processor: the time during which the processor is not synchronizing, not executing, and not measuring.

□

During a synchronization step, a processor might evaluate the precondition multiple times until it finds it to hold. To measure this, another metric counts the number of precondition evaluations in a synchronization step. Yet another metric counts the number of feature requests that have been generated for a feature. Definition 8.2 shows these two metrics.

Definition 8.2 (Count metrics). The following count metrics exist:

- *Number of precondition evaluations* of a feature request: the number of times the precondition gets evaluated in the synchronization step.
- *Number of feature requests* of a feature: the number of feature requests that have been generated for the feature.

□

The time and count metrics introduced so far are called *raw metrics*. They can be used to calculate *derived metrics*. For example, the number of feature requests can be used to calculate statistical values such as the mean and the standard deviation of all metrics defined per feature request, i.e., queue time, synchronization time, execution time, waiting time, useful time, and the number of precondition evaluations. Further, the raw metrics can be used to obtain aggregated values.

8.2 Using the metrics to diagnose issues and find solutions

The raw metrics are useful to diagnose issues in concurrent programs and find appropriate solutions. A large queue time of a feature request indicates that it takes a long time for the request to be served by the responsible processor and that therefore this processor is too busy. This can be resolved by distributing the workload onto more processors.

A large synchronization time of a feature request can be due to one of two reasons. If the feature request has a high number of precondition evaluations, then the processor is able to get the required locks, but it has problems satisfying the precondition. On the other hand, if the feature request has a low number of precondition evaluations, then the required locks are hard to obtain because it takes the processor a long time to even get to the point where it can evaluate the precondition. In the first case, one can look at the precondition to figure out why it is so hard to be satisfied. In the second case, one can introduce additional processors that replicate the roles of the congested processors.

A large execution time of a feature request is either due to a large waiting time or a large useful time. To reduce the waiting time, one can reduce the queue time, the synchronization time, or the execution time of the performed synchronous feature calls. Alternatively, one can also eliminate some of the synchronous feature calls. For instance, one can increase the concurrency granularity by synchronizing less often. To reduce the useful time, one must optimize the feature.

A large measuring time can be influenced through parameters of the measuring process (see Section 8.4 and Section 8.5). A large idle time means that a processor is not busy enough so that one can introduce more load for this processor. A large number of feature requests for a feature in combination with large queue times, synchronization times, or execution times indicates that it is especially worthwhile to optimize the feature.

8.3 From traces to metric values

This section describes how a tool can calculate the raw metrics from a *trace*, i.e., a sequence of events corresponding to one execution of a program. Each type of event is described by a class, as shown in Figure 8.2. An event has a timestamp and refers to one processor.

For a feature request r on processor p, the tool has to calculate the queue time, the synchronization time, the execution time – divided into waiting time and useful time – and the number of precondition evaluations. The following events facilitate these metrics. If r is a separate feature request that p receives from another processor g, p records a *separate feature request event*. This event contains g's identifier. As soon as p starts the synchronization step for r, p creates a *synchronization start event*. Each such event contains the identifiers of the processors whose request queue locks p requests. Each time p evaluates the precondition, it adds one *precondition evaluation event*. When p starts executing, it records an *execution start event*. If p performs a feature call to a different processor q, then p creates a *separate feature call event*. This event stores q's identifier and whether the feature call is synchronous or asynchronous. The tool can use q's identifier to

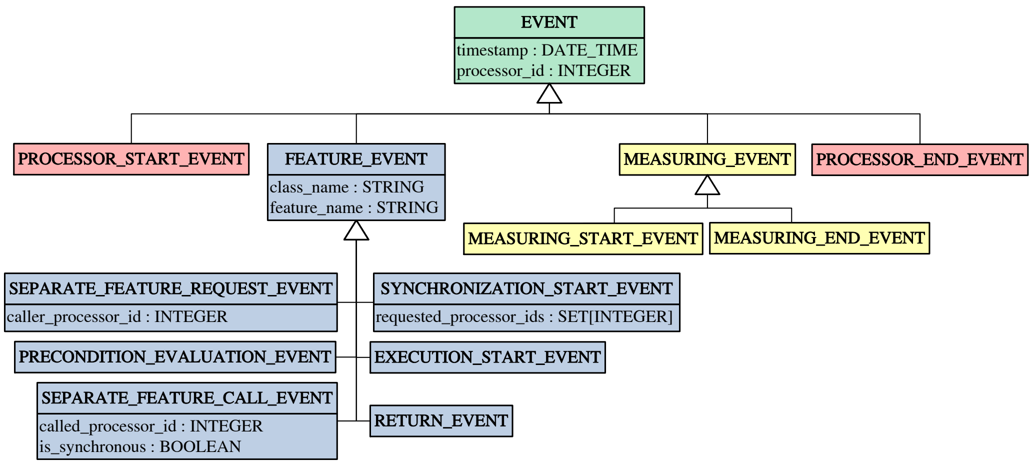


Figure 8.2: Class hierarchy for performance analysis events. The class *EVENT* is the parent of all event types. The class *FEATURE_EVENT* is the parent of all events that are related to metrics for feature requests and features; the class *MEASURING_EVENT* is the parent of all measuring event types.

find the events on q that are related to the resulting feature request by reconstructing each processor's request queue and call stack from the sequence of events. With this, the tool can link the separate feature call event on p to the related separate feature request event on q and then match this event to the remaining related events on q. After p is done with the execution, p creates a *return event*.

Figure 8.3 shows the metrics calculation for r in more detail. The notation $t_n(a)$ denotes the timestamp of the event with name n and parameter a. The notation $c_n(a)$ denotes the count of events with name n and parameter a in the trace. If r is a separate feature request, then the queue time is the time between the separate feature request event and the synchronization start event. The queue time of a non-separate feature request is zero. The synchronization time is the time between the synchronization start event and the execution start event, and the execution time is the time between the execution start event and the return event. To calculate the waiting time of r, the tool first looks for all synchronous separate feature call events that p recorded during the execution step. The time between such a separate feature call event and the matching return event on the called processor is part of the waiting time. In addition, the tool looks for synchronization start events that p recorded during the execution step; these events belong to non-separate feature calls. The time between the synchronization start event and the matching execution start event is also part of the waiting time. The useful time is the part of the execution time that is not waiting time. The number of precondition evaluations is the count of precondition evaluation events.

$$queue\_time(r) \overset{def}{=} \begin{cases} \text{if } r \text{ is a separate feature request} \\ \quad t_{synchronization\_start}(r) - t_{separate\_feature\_request}(r) \\ \text{if } r \text{ is a non-separate feature request} \\ \quad 0 \end{cases}$$

$$synchronization\_time(r) \overset{def}{=} t_{execution\_start}(r) - t_{synchronization\_start}(r)$$

$$execution\_time(r) \overset{def}{=} t_{return}(r) - t_{execution\_start}(r)$$

$$waiting\_time(r) \overset{def}{=} \sum_{i=1}^{n}(t_{return}(r_i) - t_{separate\_feature\_call}(r, w_i)) + \sum_{i=n+1}^{m}(t_{execution\_start}(r_i) - t_{synchronization\_start}(r_i))$$

where

$w_1, \ldots, w_n$ are the synchronous separate feature calls during r
$r_1, \ldots, r_n$ are the matching feature requests on the called processors
$r_{n+1}, \ldots, r_m$ are the feature requests for non-separate feature calls during r

$$useful\_time(r) \overset{def}{=} execution\_time(r) - waiting\_time(r)$$

$$number\_of\_precondition\_evaluations(r) \overset{def}{=} c_{precondition\_evaluation}(r)$$

$$number\_of\_feature\_requests(f) \overset{def}{=} c_{execution\_start}(f)$$

Figure 8.3: Metric values for a feature request r, and a feature f

For a feature f, the tool only has to calculate the number of feature requests for the feature. This number is simply the count of execution start events for f. Figure 8.3 shows this calculation.

To calculate the lifetime of a processor p, p records an event when it gets created and when it is about to be disposed. Processor p's lifetime is the time difference between these two events. To calculate the measuring time, p creates an event when a measuring action starts and another event when the action ends. Finally, the idle time of p is its lifetime minus its measuring time minus the synchronization time and execution time of each of its feature requests. Figure 8.4 shows this in more detail.

8.4 Tool overview

This section presents the tool that calculates and visualizes the metrics from Section 8.1. The tool comes with an extended SCOOP compiler to generate a *stat-*

$$lifetime(p) \overset{def}{=} t_{processor\_end}(p) - t_{processor\_start}(p)$$

$$measuring\_time(p) \overset{def}{=} \sum_{i=1}^{n}(t_{measuring\_end}(p, w_i) - t_{measuring\_start}(p, w_i))$$

where

$w_1, \ldots, w_n$ are the measuring steps on p

$$\begin{aligned} idle\_time(p) \overset{def}{=} & lifetime(p) - measuring\_time(p) - \\ & \sum_{i=1}^{n}(synchronization\_time(r_i) + execution\_time(r_i)) \\ = & lifetime(p) - \\ & measuring\_time(p) - \sum_{i=1}^{n}(t_{return}(r_i) - t_{synchronization\_start}(r_i)) \end{aligned}$$

where

$r_1, \ldots, r_n$ are the feature requests on p

Figure 8.4: Metric values for a processor p

ically instrumented executable. The compiler inserts *indirect instrumentation* code, augmenting the original source code with calls to a *performance analysis library*. With the help of the instrumentation code and the extended SCOOP runtime, the tool performs *exact monitoring*: it records the events from Section 8.3 consistently at specified execution steps.

The alternative to exact monitoring, i.e., *statistical monitoring*, is not suitable for the SCOOP performance analyzer. With statistical monitoring, the tool would record the events by periodically sampling the program state, enabling it to control the overhead by adjusting the sampling rate. However, the tool could not reliably compute intervals – a necessity for most SCOOP metrics. To compute an interval, the tool needs to know when the interval starts and when it ends. Since the tool might sample the program state at the wrong points in time, it would not collect these events reliably. Statistical monitoring is suitable for metrics that capture a single point in time. For instance, the Cilk performance analyzer [202] measures the number of idle CPU cores and the percentage of execution time used for management activities. Both of these metrics aggregate over single points in time and are therefore suitable for statistical monitoring.

To generate an event timestamp, a processor queries the system time. With a system time resolution of n milliseconds, a timestamp can have an error of $\pm n/2$ milliseconds. Hence a value for a metric consisting of a single interval can have an error of $\pm n$ milliseconds. A value for a metric consisting of m intervals, such as the waiting time metric, can have an error of $\pm(m \times n)$.

Each processor keeps its events in its own buffer, to avoid expensive synchronization actions on a central storage during the execution. A processor flushes its buffer into a file once the buffer is full. For fast lookup, it encodes in the file name the timestamps of the first and last event in the buffer. The files of all processors constitute the execution trace. To reduce overhead, the tool does not analyze this trace *online*; instead it performs a *postmortem analysis* and creates two views: the processor view and the feature view.

8.4.1 Processor view

The *processor view* has one timeline for each processor, and each timeline shows all the actions performed by one processor. Each step is presented as a bar, whose length indicates the duration of the action. The location of the bar in the timeline indicates when the action happened. The following bars exist:

- Each synchronization step is shown as a *synchronization bar*. This bar indicates the processors whose request queues got locked. It can be expanded to see the precondition evaluations as *precondition evaluation marks*.
- Each execution step is shown as an *execution bar*. This bar shows the metric values of the corresponding feature request. It can be expanded to see the feature calls: the tool shows the synchronization and execution step for each non-separate feature call and a *separate feature call bar* for each separate feature call. The separate feature call bar, whose length indicates the waiting time, is linked to the resulting execution step. Expanding the separate feature call bar shows the separate callbacks.
- Each measuring step is shown as a *measuring bar*.

For example, consider another transaction with one market, two investors, and one available share from one issuer. Each of the two investors wants first to buy the share and then to sell the share again. After the execution of this transaction each investor has two feature requests in its request queue: the first one for the *buy* feature and the second one for the *sell* feature.

```
do_transaction (
  first_investor: separate INVESTOR;
  second_investor: separate INVESTOR;
  issuer_id: INTEGER
)
```

```
    -- Make each of the two investors buy and then sell a share of the issuer on a
       market.
  do
    first_investor.buy (market, issuer_id)
    second_investor.buy (market, issuer_id)
    first_investor.sell (market, issuer_id)
    second_investor.sell (market, issuer_id)
  end
```

Figure 8.5 shows the processor view for the market application. It does not list any measuring steps because the chosen buffer size is large enough to not flush in the shown time segment. To execute the *buy* feature, both investors tried to obtain the request queue of the market while the precondition is satisfied. From the shorter synchronization bar, one can see that the second investor succeeded first. The precondition evaluation mark below the synchronization bar indicates that it evaluated the precondition only once. During its execution step, the second investor performed a feature call to the *buy* feature on the market. This separate feature call was synchronous, as indicated by the elongated separate feature call bar below the execution bar. Below the separate feature call bar, the tool shows how the market made several separate callbacks to the *id* feature.

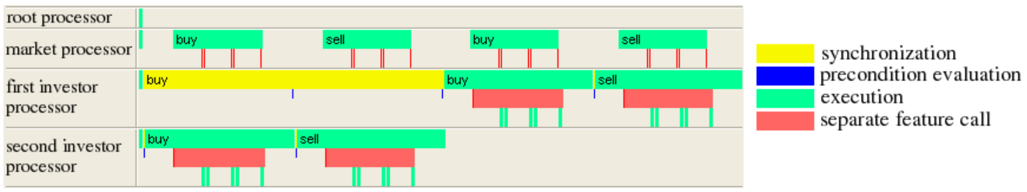
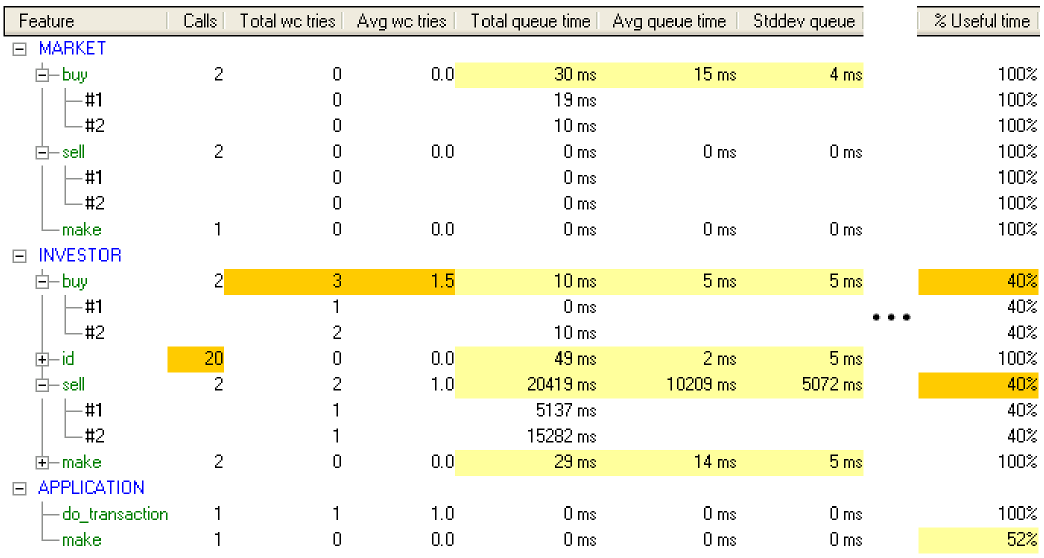


Figure 8.5: Processor view for the share market application, manually annotated with processor names and a legend

After the second investor finished the execution of the *buy* feature, it released the market's request queue lock and went straight into a second synchronization step for the *sell* feature. At this point, both investors tried to lock the market's request queue, the first investor to execute *buy* and the second investor to execute *sell*. The first investor got the lock and evaluated the precondition. However, the precondition was not satisfied since it required the availability of a share that had been sold to the second investor. Hence, the first investor released the request queue lock to let the second investor proceed with *sell*. After the execution of *sell*, the second investor released the request queue lock, letting the first investor proceed with *buy*.

8.4.2 Feature view

The *feature view* groups feature requests according to the requested features. Figure 8.6 shows an extract of the feature view for the share market application.



Feature	Calls	Total wc tries	Avg wc tries	Total queue time	Avg queue time	Stddev queue	% Useful time
MARKET							
buy	2	0	0.0	30 ms	15 ms	4 ms	100%
#1		0		19 ms			100%
#2		0		10 ms			100%
sell	2	0	0.0	0 ms	0 ms	0 ms	100%
#1		0		0 ms			100%
#2		0		0 ms			100%
make	1	0	0.0	0 ms	0 ms	0 ms	100%
INVESTOR							
buy	2	3	1.5	10 ms	5 ms	5 ms	40%
#1		1		0 ms			40%
#2		2		10 ms			40%
id	20	0	0.0	49 ms	2 ms	5 ms	100%
sell	2	2	1.0	20419 ms	10209 ms	5072 ms	40%
#1		1		5137 ms			40%
#2		1		15282 ms			40%
make	2	0	0.0	29 ms	14 ms	5 ms	100%
APPLICATION							
do_transaction	1	1	1.0	0 ms	0 ms	0 ms	100%
make	1	0	0.0	0 ms	0 ms	0 ms	52%

Figure 8.6: Extract of the feature view for the share market application. The measurements have been made on an Intel Core 2 Duo T9300 2.5 GHz CPU with 2 GB of RAM running Windows 7 SP1 (32 Bit). The system time resolution is 1 millisecond.

The first column shows a number of trees, each tree corresponding to one class. Each first level node of a tree belongs to one feature of the class; such a *feature node* stands for a group of feature requests. Each second level node belongs to one of the feature requests; such a node is a *feature request node*. The remaining columns contain the metric values in both raw and derived form.

The tool highlights some of the metric values as hotspots. A *hotspot* is a metric value that might be a good starting point to optimize the program. There are two types of hotspots: worst performers and imperfections. A *worst performer* is the worst metric value in a metric column – the largest number of feature requests, the largest number of precondition evaluations, the biggest mean of precondition evaluations, and the smallest percentage of useful time. An *imperfection* is a metric value that is not ideal, e.g., a queue time, synchronization time, or execution time above zero, or a percentage of useful time below 100%. Imperfections are not as grave as worst performers; most reasonable features are imperfect. Thus, we suggest concentrating on the worst performers when optimizing.

8.4.3 Using the tool to improve programs

This section presents a guideline on how to use the tool to improve a program. To start, one adjusts the event buffer size to have an acceptable overhead. Large buffers lead to infrequent flushes, which is useful for decreasing the time overhead at the expense of the memory overhead. Small buffers lead to frequent flushes, which is useful for decreasing the memory overhead at the expense of the time overhead. Once the overhead is acceptable, the next step is to look for hotspots in the feature view to know which feature to focus on. One then uses the processor view to find the executions of the troubled feature. Using the discussion from Section 8.2, one finds the issues and translates some of the proposed solutions to the source code. For this, one starts by opening the code of the troubled feature. The translation is not always straightforward; it requires programmers that are knowledgeable about SCOOP. These steps may need to be repeated several times as no automatic solution exists at the moment.

In Figure 8.6, for example, one can see that both features *buy* and *sell* from class *INVESTOR* have on average 40% useful time. A low percentage of useful time is the result of waiting. For this, Section 8.2 suggests to look at the synchronous feature calls in *buy* and *sell*. From the processor view, one can tell that *buy* and *sell* perform synchronous feature calls to the market, causing separate callbacks to *id*. The feature view reflects this in the number of feature requests: the *id* feature has been called 20 times. This setup decreases the performance of the program in two ways. First, the feature call to the market is synchronous. Second, the separate callbacks cause unnecessary communication. To optimize the program, one can change the *buy* feature and the *sell* feature so that the investors pass the identifier instead of the investor object itself. This change turns the synchronous feature calls into asynchronous ones. The investors do not have to wait any longer and the market does not have to perform any separate callbacks.

8.5 Evaluation

The evaluation includes an analysis of the tool's time overhead in Section 8.5.1 as well as an application of the metrics and the tool on a concurrent robotic control software in Section 8.5.2.

8.5.1 Time overhead analysis

A number of SCOOP solutions to well-known synchronization problems [60] served as subjects to measure the time overhead of the tool:

- *Barbershop problem.* A barbershop consists of a waiting room with a number of chairs and a barber room with the barber chair. When a customer arrives and the waiting room is full, the customer leaves the barbershop. When the waiting room is not full, the customer checks whether the barber is busy or not. If the barber is busy, the customer takes a seat in the waiting room; otherwise the customer gets a haircut. The experiments use ten customers, each wanting to get two haircuts, and a waiting room with five chairs.

- *Dining philosophers.* A number of philosophers sit around a round table with one chopstick between each philosopher and one bowl of food in the middle of the table. Each philosopher goes through a loop where each iteration consists of thinking, fetching the left and the right chopstick, eating, and dropping the chopsticks. The experiments use ten philosophers, each philosopher performing ten iterations.
- *Dining savages.* A tribe of savages shares a meal from a single pot. A cook is responsible to fill the large shared pot. When a savage wants to eat, he takes a serving from the pot unless the pot is empty. If the pot is empty, the savage wakes up the cook and then waits until the cook has refilled the pot. The experiments use ten savages, each savage getting five servings from the pot. The pot has a capacity of five servings.
- *Parallel workers.* An application spawns several workers and waits until each of the workers returns a result to combine the results and terminate. The experiment uses ten workers.

These programs represent various interactions that can take place in concurrent programs. Furthermore, these programs do not depend on external input, which makes them suitable for overhead measurements. However, we do not claim that these programs represent all possible interaction schemes. Figure 8.7 shows the average time overhead for the programs over different buffer sizes. The average overhead is more interesting than the overhead of a single run because of the inherent nondeterminism in concurrent programs.

Each of the programs produces a different time overhead. The reason comes from the different number of events per second: the more events a program produces per second, the higher the time overhead is. The dining savages program is the most intensive program because of the high number of exhibited interactions. The other programs show a more consistent behavior.

With growing buffer size, the time overhead of the programs decreases modulo a few bumps. The time overhead results from collecting the events in the buffers and from writing the buffers to disk. The second part contributes most to the time overhead. Since the buffer flush frequency decreases (non-strictly) with growing

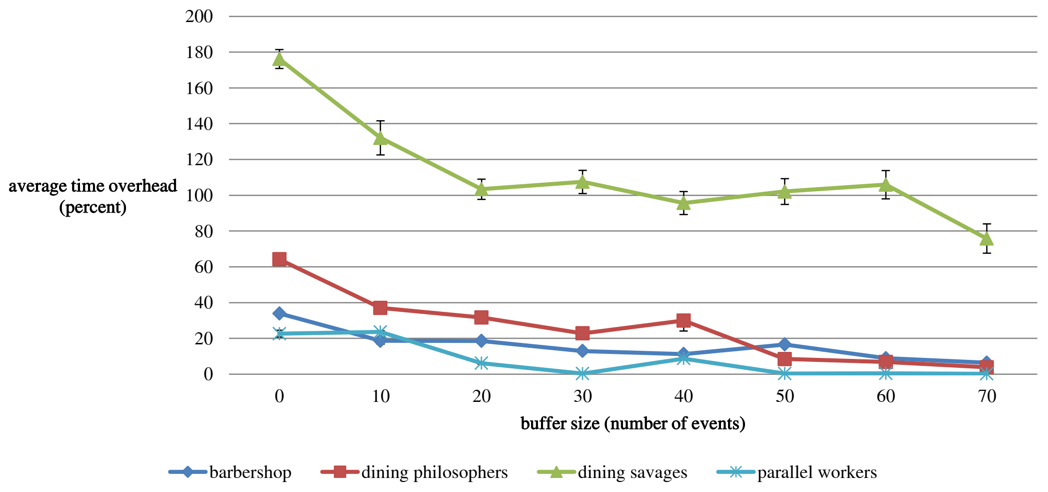


Figure 8.7: Average time overhead for different programs and varying buffer sizes. Each point reflects the average over ten runs; the bars represent the standard deviations. The baseline is the average of ten runs where the performance analyzer is disabled. A buffer size of zero means that there is no buffer; each event gets immediately written to disk after creation. The experiments have been conducted on an Intel Core 2 Duo T9300 2.5 GHz CPU with 2 GB of RAM running Windows 7 SP1 (32 Bit).

buffer size, the time overhead decreases (non-strictly) with growing buffer size as well. At some point the buffer size is large enough to hold all generated events, so that the tool can write all events at the end of the program. At this point, the overhead is minimal. One can observe this saturation especially well with the parallel workers program. The bumps can be explained with the non-strictly decreasing flush frequency. A bump is the result of an increasing time overhead due to a larger buffer size that does not get compensated by a decreasing flush frequency.

While the decentralized event collection would support an increase of processors along with an increase of CPU cores, a program with thousands of processors would require a different approach to visualization: it would not be reasonable for programmers to analyze in detail the metric values for thousands of processors; instead, programmers would require a high-level view on the metric values.

In summary, one should always adapt the buffer size according to the programs characteristics in order to minimize the time overhead. For example, one can start with a small buffer size and use the processor view to judge whether there are too many measuring steps. One can then either increase or decrease the buffer size. The biggest improvement can be expected from an increase of the buffer size from zero to a bigger value.

8.5.2 Optimizing concurrent robotic control software

This section demonstrates how the metrics and the tool can be used to optimize a concurrent robotic control software. Robotic control software is especially suited for evaluation purposes, as the SCOOP model has its main strength in expressing concurrent interactions (handling the nondeterministic arrival of events), rather than purely deterministic parallel algorithms.

Hexapod

Earlier work [184] demonstrates that SCOOP is suitable to implement concurrent robotic control software in a way that the code has a close correspondence to the behavioral specification. The hexapod robot, shown in Figure 8.8, serves as an illustration for this. The hexapod has six legs, each with three actuators. The

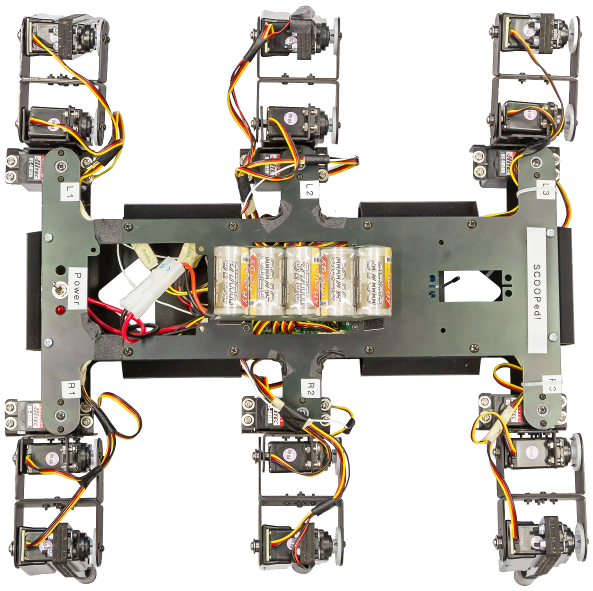


Figure 8.8: The hexapod has six legs with three degrees of freedom.

six legs are grouped into two tripods. Each tripod consists of the middle leg on one side and the outer legs on the other side, and these legs are synchronized with each other. Each tripod is equipped with sensors. A tripod's middle leg has a force sensor to determine whether the tripod is planted on the ground; the tripod is on the ground whenever the load sensor reports a weight. A tripod's middle leg also has two angle sensors to detect when the tripod reached its extreme positions in the back or in the front. The hexapod has a simple interface to retrieve the sensor values and to send commands to the actuators. This interface is accessible over an integrated ZigBee [5] wireless port.

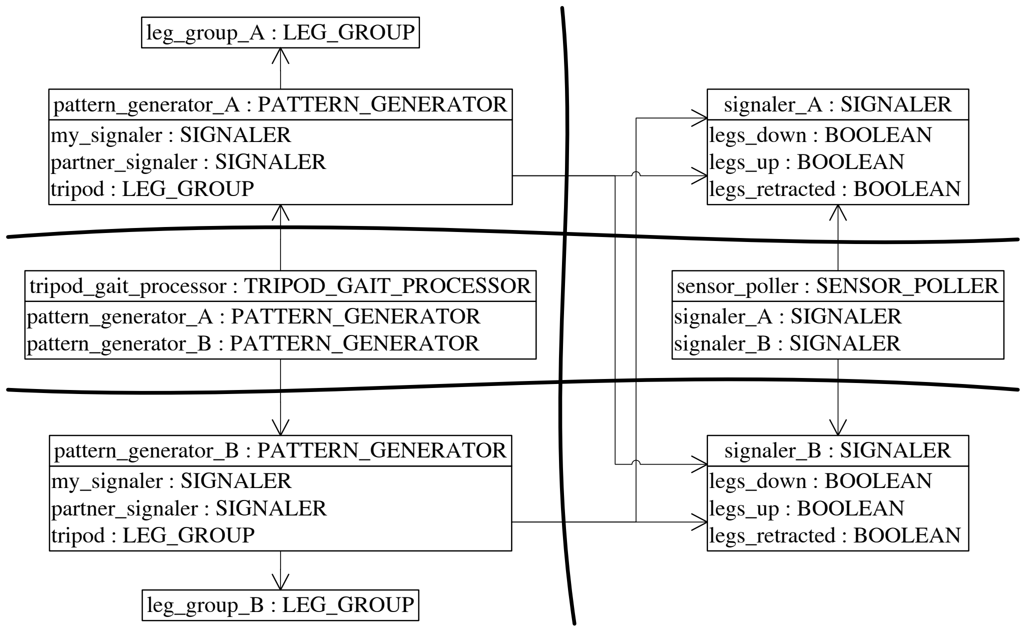


Figure 8.9: Runtime structure of the hexapod controller

The control software, shown in Figure 8.9, runs on a PC with an external ZigBee port that connects to the hexapod. At runtime, the controller has a sensor poller that periodically connects to the hexapod to retrieve the latest sensor values. The sensor poller updates two signalers, one for each tripod, that translate the sensor values into meaningful queries. For example, two such queries use the load sensor values to return whether the tripod is on the ground or not.

Two pattern generators use the signalers to control the movements of the two tripods by sending commands to the hexapod. Each tripod executes an alternating sequence of protraction and retraction. To initialize, the first tripod lifts, swings forward, and drops. Then one normal iteration begins: the second tripod lifts and swings forward while the first tripod pushes the body forward using its foot as the pivot; the second tripod drops as soon as the first tripod is done. The first tripod's action is a *protraction*. The second tripod's action is a *retraction*. In the next iteration, the two tripods change the roles.

Each pattern generator is responsible to execute the alternating sequence for one of the tripods. Each pattern generator has one feature for each phase of an iteration. Each of these features has a precondition to ensure that the tripods are in a state to start the actions. For example, a retraction of a tripod can only start when the tripod is on the ground and the other tripod is raised; otherwise the hexapod will either trip or drag its legs. The following feature shows this in detail:

```
execute_retraction (
  my_signaler: separate SIGNALER;
  partner_signaler: separate SIGNALER
)
    -- Retract the tripod.
  require
    my_signaler.legs_down
    partner_signaler.legs_up
  do
    tripod.retract (my_signaler.retraction_time)
  end
```

Optimization

The synchronization times of the pattern generator features relate to the smoothness of the hexapod's gait. When the synchronization time of a feature increases, the corresponding phase gets delayed, and the gait becomes more uneven. It is therefore important to keep the synchronization time as low as possible. The two main influences on the synchronization time are the polling frequency and the hexapod's mechanical and electronic constraints. The hexapod's constraints cannot be influenced by the controller; however, the polling frequency can be changed.

When the polling frequency increases, the controller knows about the hexapod's state sooner, which reduces the synchronization time. On the other hand, the polling frequency should be kept as low as possible to reduce the CPU time of the sensor poller and to reduce the network traffic between the controller and the hexapod. The optimal polling frequency is therefore a trade-off.

The synchronization time metric can be used to find the optimal polling frequency. For this purpose it is necessary to introduce a delay into the code of the sensor poller. One can then use the tool to measure the synchronization times of the pattern generator features and reduce the polling frequency until the synchronization times increase. Figure 8.10 shows the tool's reported average total synchronization time and the average polling frequency for different delays in the sensor poller. The progression of the average polling frequency confirms that increasing the delay indeed reduces polling. The course of the average total synchronization time shows that synchronization becomes slower as the polling gets reduced. It is however interesting to see that the first 20 milliseconds of delay do not have a big impact on synchronization. One can therefore reduce the polling

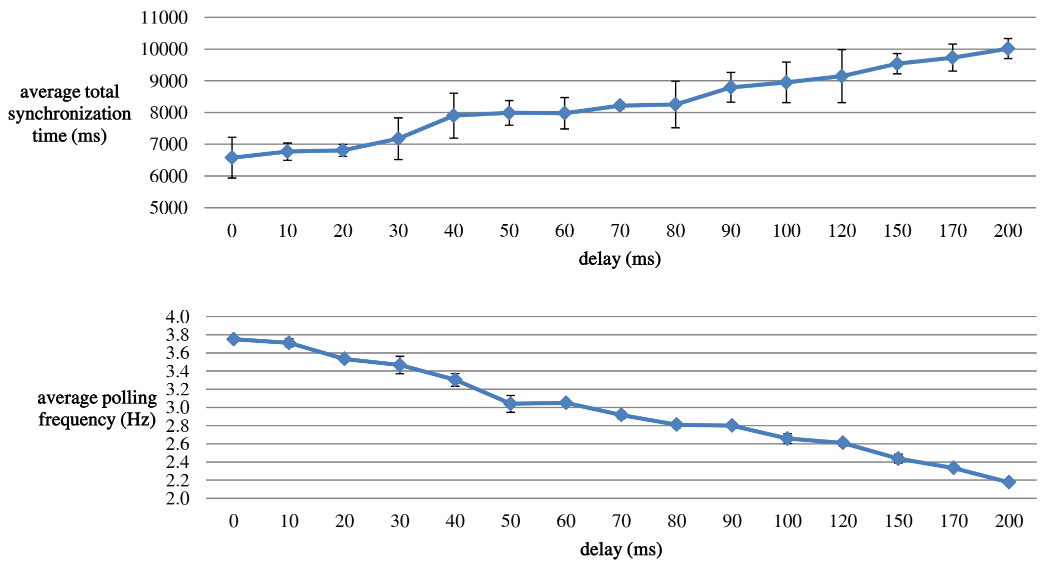


Figure 8.10: Average total synchronization time and average polling frequency for varying polling delays. Each point reflects the average over three walks, each consisting of several iterations; the bars represent the standard deviations.

frequency with a minimal impact on the hexapod's gait.

One concern remains to be answered: the tool is creating overhead, and thus it is not immediately clear whether the results are transferable to executions without the tool. In general, the overhead could cause a fundamentally different schedule. In the case at hand, however, only few schedules are possible. In the beginning, either tripod can start. Due to the symmetry of the hexapod (3 legs on each side), both choices do not fundamentally change the schedule. Henceforth, the preconditions ensure that the tripods proceed in the predefined order. For instance, the precondition of *execute_retraction* ensures that the retraction of one tripod only starts after the other tripod has lifted (see *partner_signaler.legs_up*). Similarly, another precondition ensures that one tripod can only drop after the other tripod has retracted. In fact, the only remaining scheduling ambiguity is whether one tripod starts swinging forward before the other tripod starts retracting, or the other way around. This choice, however, also happens without the tool and does not change the essence of the schedule.

The tool's overhead, however, causes an additional delay in the sensor poller. Due to this additional delay, the curves in Figure 8.10 must be shifted to the right to transfer the results to executions without the tool. Furthermore, the overhead coming from the synchronization start events, the precondition evaluation events, and the execution start events affect the average total synchronization time. Hence, the curve with the average total synchronization time must be shifted to the bottom. Note that these two shifts are conservative in the sense that the above conclusions still hold on executions without the tool.

The optimization of the controller is not limited to the sensor poller. The number of feature requests also indicates that the controller performs a lot of unnecessary separate feature calls such as *my_signaler.retraction_time* in the feature *execute_retraction*. As this feature call returns a value that is constant over an execution, the value can be cached in the pattern generators, which eliminates this and other similar calls. The tool is also helpful to visually validate the gait by inspecting the processor view for the controller.

8.6 Applicability to other concurrency models

The proposed metrics not only apply to SCOOP, but they can also be adapted to concurrency models that share SCOOP's core ideas, i.e., assignment of an object to one autonomous thread of control along with a request queue and atomic locking with preconditions that have wait semantics. One such model is Cameo [36], where each object has its own thread of control and an associated request queue for feature requests from other objects. All feature calls are synchronous unless the async keyword indicates otherwise. To perform a remote feature call, a client adds a feature request to the supplier's request queue. The behavior of a local feature call depends on the nature of the call: if the call is synchronous, the object processes the call immediately; if the call is asynchronous, the object adds a feature request to its own request queue. Before executing a feature, an object locks the actual arguments for the duration of the execution. It also delays the execution until the feature's wait condition is satisfied. When one of its suppliers is in need of its locks, it temporarily passes its locks to the supplier. The object implicitly obtains a *lazy lock* on a supplier that is not a locked actual argument. The lazy lock lasts just for the duration of the feature call on the supplier. A lazy lock is semantically equivalent to a synchronous feature call to a local wrapper with the same precondition as the called feature.

To adapt the proposed metrics to Cameo, one must translate the terminology. Separate feature calls become remote feature calls, non-separate feature calls become local feature calls, and processors become objects. In addition to these terminological changes, one must make a number of semantic changes. The *queue time* metric can be reused with a change: the queue time of a local feature call is not always zero; for asynchronous local feature calls, it is the time between the feature call and the beginning of the feature application. The *synchronization time* and *execution time* metrics can be used without a change; however, a lazy lock must be treated as an implicit local feature call with an own synchronization and execution time. Lastly, the *waiting time* metric must exclude asynchronous local feature calls because an object does not wait for itself to process such a call.

8.7 Related work

While we are the first to propose performance metrics for SCOOP, performance metrics have been proposed for other programming models. This section discusses these metrics and the tools that compute them. It focuses on the most interesting metrics and thus does not discuss elementary metrics such as the execution time, number of invocations, IO activity, or memory usage.

The Intel VTune Amplifier XE [103] is a performance analysis tool for multi-threaded CPU and GPU applications. It uses statistical and exact monitoring. The tool calculates various architecture-specific metrics such as CPU core idleness and the number of wrong branch predictions. It also has metrics that measure waiting of threads and locking of synchronization objects. The tool is supported by a Performance Monitoring Unit that is part of Intel CPUs. AMD CodeXL [10] provides similar functionality for programs running on AMD CPUs. Just as the VTune Amplifier XE, the SCOOP performance analyzer keeps track of waiting on resources; it does so with the synchronization time and the number of precondition evaluations metrics. However, it does not record idle CPU cores and other architecture-specific values. Its metrics are all on the level of the programming model to make it easier to relate the metrics to the code.

Wolf and Mohr [158, 217] introduce KOJAK to analyze the performance of MPI [146] and OpenMP [172] programs using exact monitoring. KOJAK offers a number of metrics to measure communication inefficiencies, lock and barrier waiting times, idleness, and runtime overhead. For instance, the *late sender* metric represents the time wasted when a receiver executes a blocking receive operation before the sender starts. The metrics relate directly to concepts of the programming model and are thus helpful to find issues in programs. The communication inefficiencies metrics are similar to the queue time in the SCOOP performance analyzer because both represent the gap between a sender and a receiver. The lock waiting time and the barrier waiting time metrics have a similar purpose than the synchronization time metric. These similarities are not surprising because most concurrent programming models have notions for communication and synchronization, and these notions must be reflected in any set of model-specific metrics.

Geimer et al. [78] evolve KOJAK into SCALASCA. Compared to KOJAK, SCALASCA can identify wait states even in executions with a large number of processes. It supports incremental performance analysis since it can produce either a simple online profile or a complete postmortem analysis. SCALASCA extends KOJAK's metrics with architecture-specific metrics and offers many new visualizations. Lorentz et al [137] complements SCALASCA with further visualizations and a metric that measures the accumulated waiting time caused by a delay.

Vampir by Brunst, Knüpfer, et al. [39, 123] also has its roots in KOJAK. Vampir measures communication metrics and architecture-specific metrics, including metrics that measure energy consumption. The HPCTOOLKIT [2] by Adhianto et al. is another performance analyzer for MPI and OpenMP. It uses statistical monitoring to compute architecture-specific metrics. Shende and Malony [139, 196] introduce Tau, a performance analyzer for many programming models. Tau uses statistical or exact monitoring to measure architecture-specific metrics.

Tallnet et al. [202] propose a performance analysis tool for Cilk [24] using statistical monitoring. The tool has two main metrics: parallel idleness and parallel overhead. *Parallel idleness* of a program unit is the number of CPU cores that are idle during the execution. *Parallel overhead* of a program unit is the percentage of execution time that is used for management activities. These metrics can be used to diagnose performance problems of a program unit. If the program unit has both low parallel idleness and low parallel overhead, then the parallelization is optimal. If only the parallel idleness is high, then one should refine concurrency granularity. If only the parallel overhead is high, then one should coarsen the concurrency granularity. If both metrics are high, then one should change the strategy; for instance, one can use a combination of data and task parallelism instead of just one of them. These metrics are well-suited for Cilk programs because concurrency is triggered implicitly by the runtime; thus it is indeed important to audit the runtime by measuring the CPU core utilization and the imposed overhead. These measurements are less important in SCOOP because programmers trigger concurrency explicitly. This difference has another implication. Since Cilk programmers do not need to know in detail how the runtime executes, the Cilk performance analyzer shows the metric values in a call graph. In contrast, the SCOOP performance analyzer distributes the metric values over the timelines to show the explicitly triggered concurrent executions.

Su et al. [201] present a performance analysis tool for programs based on the Partitioned Global Address Space (PGAS) model [6]. A PGAS system has a global memory address space that is partitioned among nodes; all nodes can access all partitions. The tool has an interface that PGAS systems implement to provide the necessary measurements. The tool offers metrics to measure the computational and storage load of nodes as well as the communication between different nodes. These metrics are well-suited for PGAS systems because they have a direct correspondence to the model. These metrics are also useful to describe the load and communication of SCOOP processors. However, they do not capture other aspects of the SCOOP model such as synchronization.

Miller et al. [152] present Paradyn, a performance analysis tool for MPI-based [146] and multi-threaded programs. The tool uses dynamic instrumentation to perform statistical monitoring on demand and is therefore suitable to analyze long-running programs. It supports a number of metrics to analyze communication and synchronization. Paradyn relies on Dyninst [95] to dynamically instrument executables. The SCOOP performance analyzer uses static instrumentation due to the lack of support in SCOOP.

Hollingsworth et al. [93] present a quantitative technique to compare the effectiveness of concurrent performance metrics in guiding programmers. The True Zeroing approach defines a reference metric. For each method, the effect on total execution time of removing the method from the application. To remove the method, the authors first execute the application on a given input and record the state changes made by the method. They then replace each call to the method with a number of assignments that make the same changes to the state. To perform the comparison, they calculate for each method the value of the reference metric and the value of all other metrics. For each metric, they determine the suggested order in which the methods should be optimized. For each order, they calculate the normalized differences between the metric values of the ordered methods. Based on this, they calculate a correlation coefficient between each metric to be compared and the reference metric. The authors use this technique to compare the metrics of the following approaches:

- The IPS-2 approach [153] computes for each method the useful CPU time.
- The NPT approach [11] calculates for each method the time the executing process is doing useful work divided by the number of processes doing useful work at the same time. The approach suggests optimizing the method with the longest execution time and the largest number of processes that wait while the method is being executed.
- The critical path approach [220] determines the critical path and then sums up the useful CPU time for each method on the critical path.
- The Logical Zeroing approach [153] compares for each method the total execution time of the original critical path with the total execution time of the alternative critical path, where the duration of the method is set to zero. This gives an approximation of how much the total execution time will change when the method is optimized. The approach does not consider that the optimization might cause event reordering.
- The Slack approach [94] first determines the critical path. For each method on the critical path, the *slack value* indicates how much the method can be improved before the critical path will change. The motivation is that an improvement of the longest executing method on the critical path might not

> significantly improve the total execution time because the critical path might change to a path that is slightly shorter. The approach does not work for applications where all paths are comparably long. It also ignores situations in which a method on the critical path is also on a secondary path and hence an improvement of the method will reduce the total execution time of both paths.

The authors conclude that there is no single universal metric. Performance analysis tools must support multiple metrics and assist in the selection. The metrics can be applied to most concurrency models. This makes them general, but also less expressive. For example, they are not designed to find synchronization issues, as done in the analysis of the hexapod (see Section 8.5). The SCOOP performance analyzer does not perform critical path analysis; however, it would be straightforward to calculate the critical path from the processor view.

Snelick [198] presents S-Check, a performance analysis tool to determine the effect of changes in the code without actually performing the changes. S-Check is based on synthetic perturbation screening. This technique assumes that if a code segment is sensitive to perturbations, then a comparable improvement will boost the performance in a similar way. In a first step, S-Check assists in picking code lines that are suspected bottlenecks. Then S-Check inserts artificial delays at the specified code lines in a way that each delay can be enabled or disabled; each delay combination generates a new version of the program. S-Check executes some of the versions and records the total execution times. This results in a total execution time for some of the delay combinations. Based on this, S-Check fits a polynomial function whose input is a delay combination and whose output is the total execution time for the delay combination. With this function, S-Check extrapolates the effect of each delay. The approach suggests optimizing the line of code whose delay has the largest effect. To find out whether the basic assumption is valid, this approach could be combined with the Slack approach.

Cai et al. [44] develop a monitor for programs written in occam [195], a concurrent programming language based on CSP [91]. The monitor minimizes the *probe effect* (the alteration of the system's behavior due to the measurement). It guarantees that in each execution the events are identical regardless of the monitor's presence. The monitor uses a variation of logical clocks [128] to order events partially. Every process has one logical clock that records the execution time minus the delay caused by the monitor. Whenever a process wants to pick one channel from a set of available ones, the monitor delays the decision until it finds the communication partner that has a matching communication event with the lowest logical clock timestamp. To apply this approach to SCOOP, the monitor would only allow a processor to obtain a lock when it is sure that no other processor with an earlier logical clock value will try to obtain the lock. However, in SCOOP the set of processors that will try to obtain a lock is generally unknown. Hence, this approach is not directly applicable to SCOOP.

8.8 Discussion

Motivations for making a program concurrent include optimizing the performance, improving the reliability, bringing convenience to users, and profiting from the modeling power of concurrent programming languages. To assess concurrent programs with respect to the first goal, performance analysis is critical. New models, such as SCOOP, often have an execution semantics that differs considerably from established models. These models thus require new tools with performance analysis metrics that are tailored their semantics.

This chapter presented a tool to find performance issues in SCOOP programs, using a number of novel metrics that are specific to the SCOOP model. For example, one of the newly-introduced metrics is the synchronization time metric. The synchronization time metric represents the time it takes to obtain the request queue locks and to satisfy the precondition. This metric relates directly to the smoothness of the robot's gait when applied to the hexapod control, proving its usefulness in optimizing concurrent software.

In a future version of the tool, the processor view can show the critical path along with a hotspot based on Slack values [94], and also show when a processor unlocks itself. The precondition evaluation marks can show a breakdown of the precondition, allowing programmers to see the parts of the precondition that were satisfied and the ones that could not be satisfied. The feature view can be extended with the Logical Zeroing analysis [153] and the sensitivity analysis [198]. A further view can show for each processor how often its request queue lock has been requested and in which contexts the lock has been requested. A hotspot can show the processor whose request queue lock has been requested most often.

For a future distributed implementations, it will be necessary to also measure the communication impact. The timestamps of the events could be based on logical clocks such as Lamport clocks [128] or vector clocks [141] for the ordering of events and synchronized clocks using the Network Time Protocol [155] for the approximation of durations. In practice, the Network Time Protocol could be precise enough to order the events from different nodes.

9 Conclusion

This thesis presented a method for developing concurrency models in a systematic way and applied the method to clarify and extend an object-oriented concurrency model, SCOOP. The presented method relies on executable formal specifications to conduct testing prior to developing compiler and runtime support. The benefits of using this method was demonstrated on SCOOP. Prior to this thesis, SCOOP was defined informally in [150, 164], but no complete and comprehensive formal specification existed. This thesis used the proposed method to clarify SCOOP and extend SCOOP with two new mechanisms - asynchronous exceptions and fast and safe data sharing. In addition to clarification and extension of SCOOP, this thesis resulted in a comprehensive executable formal specification of SCOOP implemented in Maude; the executable formal specification can serve as a reference and starting point for future model extensions.

9.1 Defense of the thesis statement

Presentation of a method would not be complete without a thorough evaluation. This thesis applied the prototyping method to SCOOP, and the process revealed several clarifications. Some of these clarifications could only be found through a comprehensive study of SCOOP. For example, Clarification 7.1 could only be revealed by understanding both passive processor and import operation for expanded object fully. In general, the clarifications minimize the risk of the runtime and complier suffering from semantic flaws. A complete list of clarifications are as follows:

- Clarification 3.1. Once routines must not have a result type with an explicit processor specification.

- Clarification 3.2. Each processor defines a lock for its request queue and a lock for its call stack.

- Clarification 3.3. The deep import operation atomically locks the handlers of all reachable separate objects, which is too costly. The import operation is a better fit, but requires a convention: in imported object structures, invariants must not rely on object identities.

- Clarification 3.4. Let q be a processor about to import an object o handled by a different processor p; o has a non-separate once function or once procedure f. In case f is not fresh on p and fresh on q, f remains fresh on q.

- Clarification 3.4. In imported object structures, non-separate once functions must compute results that satisfy the object structure's invariants when recomputed on any processor.

- Clarification 3.5. During a feature call, the client p passes its locks to the supplier q in two cases: (1) p holds a lock on a processor handling an argument of attached reference type; or (2) the feature call is a separate callback. In both cases, p passes all its locks, including its call stack lock.

- Clarification 3.6. During a feature call, the client must copy or import expanded objects to a supplier.

- Clarification 3.7. The postcondition of a feature f must be evaluated by the processor executing f.

- Clarification 5.1. A separate callback is characterized as: the supplier has passed its locks, and the client holds the supplier's call stack lock.

- Clarification 5.2. Just after its first execution in a context, a once routine f is not fresh and *not stable*; after its execution, f becomes not fresh and *stable*. The processor that executes f for the first time is the *stabilizer*. Only the stabilizer can execute f while it is not stable. If f is a separate once function, the stabilizer must only call f on non-separate targets; furthermore, it must ensure that the once result is meaningful before any calls with lock passing.

- Clarification 5.3. During a query call, a processor passes all its locks.

- Clarification 5.4. A supplier p of a command call must only synchronize with the client q, if q has passed any locks to p.

- Clarification 6.1. A precondition of a feature f is a correctness condition if and only if (1) the call to f is synchronous and (2) the client claims the request queue lock or the call stack lock for every target in the precondition. Every other precondition is a wait condition.
- Clarification 6.2. Before releasing the request queue locks of processors $\{g_1, \ldots, g_n\}$ in safe mode, a processor p asks each $g \in \{g_1, \ldots, g_n\}$ to send a notification after finishing the issued workload; for all $i \in \{1, n-1\}$, p waits for g_i's notification before asking g_{i+1}.
- Clarification 6.3. Before unlocking the request queue of a supplier q after a creation call in safe mode, the client p must query q for any failures and wait for the notification.
- Clarification 7.1. If p returns an expanded object with reference r from a passive processor to itself or another passive processor, it must import r to that processor.

The list of clarifications is an evidence that the prototyping method is beneficial to clarify a concurrency model before implementing compiler and runtime support. Without these clarifications, the compiler and runtime developers would have to debug many lines of code to resolve the flaws in the model. With the prototyping method, the semantics of the model can be studied in a purer environment before its compiler and runtime support are introduced. Although executable specifications suffer from the same criticism of principle as program tests in that the approach can at best give partial reassurance, never a full guarantee, the results reported here suggest that executable specifications are, in practice, a realistically usable and fruitful path to the verification of semantic models. Having successfully demonstrated the benefits of the method on an extensive concurrency model, we believe that our prototyping method will also prove its usefulness in the development of other models.

9.2 Future work

The work on this thesis has opened up several directions for future work. The possible directions vary from automatic testing of the executable formal specification to introducing teaching instrument and extending SCOOP itself.

- *Testing framework.* A tool can use the executable formal specification to run test program and check them automatically against a specified outcome. The tool can use Maude strategies (see Section 4.4.2) to test different scheduling algorithms.

- *Compiler and runtime conformance checking*. A tool can run a compiled test programs and run the same program using the executable formal specification; it can then check whether the two executions conform to each other.

- *Teaching instrument*. Similar to Eiffel// [14] (see Section 5.6), a front-end for SCOOP could enable students to develop a program and navigate through different executions. The front-end translates the SCOOP program into Maude syntax, uses Maude strategies to realize the executions, and visualizes the Maude output. Advanced students can even explore variants of the SCOOP semantics by modifying the executable formal specification; no lengthy recompilation of the compiler and the runtime is necessary.

- *Performance simulation*. Each operation can be annotated with timing data. Maude can then compute the values for the metrics in Section 8.1 under various executions without any compiler or runtime support, thus complementing the performance analysis tool from Section 8.1. The effectiveness of such performance simulation tools has been shown by Hoefler et al. [92] and others.

- *Further extensions to SCOOP*. The prototyping method can be used to extend SCOOP further with:

  - *Lock delegation* (see Section 6.8). A client p can delegate its obtained locks to another client q; after delegation, p asynchronously continues without the delegated locks.
  - *Shared read locks* (see Section 7.7). Multiple clients can simultaneously read objects on a passive processor.
  - *Duel mechanism* [150]. A processor can challenge another processor for its locks.
  - *Garbage collector* [151]. The system collects unused processors and objects. The garbage collector in the current executable formal specification only collects objects.
  - *Deadlock detection*. The system detects deadlocks due to unavailable locks and unsatisfied wait conditions. This mechanism complements West et al.'s static deadlock prevention approach for SCOOP programs [215].

- *Eiffel coverage*. While the executable formal specification covers all operational aspects of SCOOP, it does not cover the entirety [105] of Eiffel. In the future, it could support more Eiffel aspects such as inheritance, genericity, agents, object tests, and entity initialization.

- *Application to other concurrency models.* The prototyping method can be used to develop concurrency models other than SCOOP.

9.3 Final remarks

The thesis presented a prototyping method for a systematic development of concurrent models and demonstrated its effectiveness on SCOOP. By applying the prototyping method, it clarified SCOOP and added two much needed mechanisms - asynchronous exceptions and efficient data sharing. In terms of exception handling, this thesis introduced the accountability framework with concepts to describe asynchronous exception mechanisms and presented a classification of approaches along with a discussion of their strengths and weaknesses. It used the accountability framework and the classification to derive a mechanism for asynchronous exceptions in SCOOP. In terms of data-sharing, this thesis further extended SCOOP's message-passing foundation with passive processors. Passive processors support safe data sharing and reduce execution time by several orders of magnitude on data-intensive parallel programs. Lastly, the thesis presented SCOOP-specific performance analysis metrics to enable programmers use SCOOP more effectively. The performance analyzer diagnoses performance bottlenecks in SCOOP programs, easing the analysis and improvement of programs. The performance analysis metrics, the clarifications, and the new mechanisms extend SCOOP significantly. Because of these advances, we feel confident to add our proposal to the record started by Meyer's SCOOP_97 [150] and continued by Nienaltowski's SCOOP_07 [164] by naming it SCOOP_14.

Bibliography

[1] Erika Ábrahám. *An Assertional Proof System for Multithreaded Java – Theory and Tool Support*. PhD thesis, University of Leiden, 2005.

[2] Laksono Adhianto, Sinchan Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John M. Mellor-Crummey, and Nathan R. Tallent. HPC-TOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[3] Eric Allen, David Chase, Joe Hallet, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification. Technical report, Sun, 2008.

[4] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Guy L. Steele Jr. Object-oriented units of measurement. In *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 384–403, 2004.

[5] ZigBee Alliance. ZigBee specification. Technical report, ZigBee Alliance, 2012.

[6] George Almasi. Partitioned global address space (PGAS) languages. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, page 1465. Springer, 2011.

[7] Musab AlTurki and José Meseguer. Real-time rewriting semantics of Orc. In *International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 131–142, 2007.

[8] Musab AlTurki and José Meseguer. Reduction semantics and formal analysis of Orc programs. *Electronic Notes in Theoretical Computer Science*, 200(3):25–41, 2008.

[9] Musab AlTurki and José Meseguer. Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis. In *International Workshop on Rewriting Techniques for Real-Time Systems*, pages 26–45, 2010.

[10] AMD. CodeXL. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/codexl/, 2013.

[11] Thomas E. Anderson and Edward D. Lazowska. Quartz: a tool for tuning parallel program performance. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, 1990.

[12] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[13] Volkan Arslan and Bertrand Meyer. Asynchronous exceptions in concurrent object-oriented programming. In *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-Like Languages*, pages 62–70, 2006.

[14] Isabelle Attali, Denis Caromel, Sidi O. Ehmety, and Sylvain Lippi. Semantic-based visualization for parallel object-oriented programming. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 421–440, 1996.

[15] Rafael Bohrer Ávila, Caciano Machado, and Philippe O. A. Navaux. Message-passing over shared memory for the SECK programming environment. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 590–595, 2002.

[16] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. Grid computing: Software environments and tools. chapter Programming, Composing, Deploying for the Grid, pages 205–229. Springer, 2006.

[17] Arnaud Bailly. Formal semantics and proof system for SCOOP. `http://se.inf.ethz.ch/`, 2004.

[18] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[19] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: from C++11 to

POWER. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 509–520, 2012.

[20] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 55–66, 2011.

[21] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):269–804, 2004.

[22] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2010.

[23] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens, and Susmit Sarkar. Nitpicking C++ concurrency. In *International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 113–124, 2011.

[24] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[25] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *IEEE Symposium on Computer Arithmetic*, pages 107–115, 2013.

[26] Greg Bollella, James Gosling, Benjamin M. Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[27] Viviana Bono, Chiara Messa, and Luca Padovani. Typing copyless message passing. In *European Symposium on Programming*, pages 57–76, 2011.

[28] Egon Börger, Nicu G. Fruja, Vincenzo Gervasi, and Robert F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2–3):235–284, 2005.

[29] Egon Börger and Robert F. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[30] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4:35–50, 1996.

[31] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. CENTAUR: The system. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1988.

[32] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1-2):35–132, 2000.

[33] Martin Bravenboer, Karl T. Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52–70, 2008.

[34] Ron Brightwell. Exploiting direct access shared memory for MPI on multicore processors. *International Journal of High Performance Computing Applications*, 24(1):69–77, 2010.

[35] Phillip J. Brooke and Richard F. Paige. Exceptions in concurrent Eiffel. *Journal of Object Technology*, 6(10):111–126, 2007.

[36] Phillip J. Brooke and Richard F. Paige. Cameo: an alternative model of concurrency for Eiffel. *Formal Aspects of Computing*, 21(4):363–391, 2009.

[37] Phillip J. Brooke, Richard F. Paige, and Jeremy L. Jacob. A CSP model of Eiffel's SCOOP. *Formal Aspects of Computing*, 19(4):487–512, 2007.

[38] Roberto Bruni and José Meseguer. Generalized rewrite theories. In *International Colloquium on Automata, Languages and Programming*, pages 252–266, 2003.

[39] Holger Brunst, Daniel Hackenberg, Guido Juckeland, and Heide Rohling. Comprehensive performance tracking with Vampir 7. In *International Parallel Tools Workshop*, pages 17–29, 2009.

[40] Peter A. Buhr. μC++ annotated reference manual. Technical report, University of Waterloo, 2013.

[41] J. Mark Bull, James P. Enright, Xu Guo, Chris Maynard, and Fiona Reid. Performance evaluation of mixed-mode OpenMP/MPI implementations. *International Journal of Parallel Programming*, 38(5–6):396–417, 2010.

[42] Darius Buntinas, Guillaume Mercier, and William Gropp. Data transfers between processes in an SMP system: Performance study and application to MPI. In *International Conference on Parallel Processing*, pages 487–496, 2006.

[43] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Computing*, 33(9):634–644, 2007.

[44] Wentong Cai and Stephen John Turner. An approach to the run-time monitoring of parallel programs. *The Computer Journal*, 37(4):333–345, 1994.

[45] Roy H. Campbell and Brian Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.

[46] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Supercomputing 2000*, 2000.

[47] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[48] Denis Caromel and Guillaume Chazarain. Robust exception handling in an asynchronous environment. In *Workshop on Exception Handling in Object-Oriented Systems*, 2005.

[49] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer, 2004.

[50] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The java memory model: Operationally, denotationally, axiomatically. In *European Symposium on Programming*, pages 331–346, 2007.

[51] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[52] Martin J. Chorley and David W. Walker. Performance analysis of a hybrid mpi/openmp application on multi-core clusters. *Journal of Computational Science*, 1(3):168–174, 2010.

[53] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework*. Springer, 2007.

[54] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. *Electronic Notes in Theoretical Computer Science*, 4:65–89, 1996.

[55] Edward G. Coffman, Melanie J. Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[56] Michael Compton and Richard Walker. A run-time system for SCOOP. *Journal of Object Technology*, 1(3):119–157, 2002.

[57] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[58] Christophe Dony, Christelle Urtado, and Sylvain Vauttier. Exception handling and asynchronous active objects: Issues and proposal. In *Advanced Topics in Exception Handling Techniques*, pages 81–100, 2006.

[59] Hans Rohnert Frank Buschmann Douglas Schmidt, Michael Stal. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[60] Allen B. Downey. *The Little Book of Semaphores*. Green Tea Press, 2nd edition, 2005.

[61] Ian N. Dunn and Gerard G.L. Meyer. Parallel QR factorization for hybrid message passing/shared memory operation. *Journal of the Franklin Institute*, 338(5):601–613, 2001.

[62] Eiffel Software. EiffelStudio. `http://www.eiffel.com/`, 2013.

[63] Steven Eker, Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. Deduction, strategies, and rewriting. *Electronic Notes in Theoretical Computer Science*, 174(11):3–25, 2007.

[64] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2004.

[65] Chucky Ellison. *A Formal Semantics of C with Applications*. PhD thesis, University of Illinois, 2012.

[66] Ericsson. Erlang/OTP system documentation. Technical report, Ericsson, 2012.

[67] ETH Zurich. EVE. `https://trac.inf.ethz.ch/trac/meyer/eve/`, 2013.

[68] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roşu. Formal analysis of Java programs in JavaFAN. In *International Conference on Computer Aided Verification*, pages 501–505, 2004.

[69] Azadeh Farzan, José Meseguer, and Grigore Roşu. Formal JVM code analysis in JavaFAN. In *Algebraic Methodology and Software Technology*, pages 132–147, 2004.

[70] Matthias Felleisen, Robert B. Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[71] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[72] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[73] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–385, 1996.

[74] Peter Fritzson, Adrian Pop, David Broman, and Peter Aronsson. Formal semantics based translator generation and tool development in practice. In *Australian Software Engineering Conference*, pages 256–266, 2009.

[75] Nicu G. Fruja. Specification and implementation problems for C#. In *Abstract State Machines*, pages 127–143, 2004.

[76] Oleksandr Fuks, Jonathan S. Ostroff, and Richard F. Paige. SECG: The SCOOP-to-Eiffel Code Generator. *Journal of Object Technology*, 3(10):143–161, 2004.

[77] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Annual ACM Symposium on Principles of Programming Languages*, pages 52–66, 1985.

[78] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The SCALASCA performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[79] David Gelernter, Nicholas Carriero, Sarat Chandran, and Silva Chang. Parallel programming in Linda. In *International Conference on Parallel Processing*, pages 255–263, 1985.

[80] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[81] Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. In *Annual ACM Symposium on Theory of Computing*, pages 89–94, 1978.

[82] James Gosling, Bill Joy, Guy L. Steele Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley, 2013.

[83] Richard L. Graham and Galen M. Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *European PVM/MPI Users Group Meeting*, pages 130–140, 2008.

[84] Olivier Gruber and Fabienne Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In *European Conference on Object-Oriented Programming*, pages 281–301, 2013.

[85] Yuri Gurevich. Specification and validation methods. chapter Evolving algebras 1993: Lipari guide, pages 9–36. Oxford University Press, 1995.

[86] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 343(3):370–412, 2005.

[87] Jens Gustedt. Data handover: Reconciling message passing and shared memory. In *Foundations of Global Computing*, 2005.

[88] Per B. Hansen. The programming language Concurrent Pascal. *IEEE Transaction on Software Engineering*, 1(2):199–207, 1975.

[89] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[90] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[91] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[92] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. LogGOPSim - simulating large-scale applications in the LogGOPS model. In *ACM International Symposium on High Performance Distributed Computing*, pages 597–604, 2010.

[93] Jeffrey K. Hollingsworth and Barton P. Miller. Parallel program performance metrics: A comparison and validation. In *ACM/IEEE Conference on Supercomputing*, pages 4–13, 1992.

[94] Jeffrey K. Hollingsworth and Barton P. Miller. Slack: a new performance metric for parallel programs. Technical report, University of Wisconsin Madison, 1994.

[95] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High Performance Computing Conference*, pages 841–850, 1994.

[96] Daniel T. Huang and Ronald A. Olsson. An exception handling mechanism for SR. *Computer Languages, Systems & Structures*, 15(3):163–176, 1990.

[97] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[98] Yuuji Ichisugi and Akinori Yonezawa. Exception handling and real time features in an object-oriented concurrent language. In *Concurrency: Theory, Language, and Architecture*, pages 91–109, 1991.

[99] Information-technology Promotion Agency. Ruby. Technical report, Information-technology Promotion Agency, 2010.

[100] Insititue of Electrical and Electronics Engineers. IEEE standard for the Scheme programming language. Technical Report 1178-1990, Insititue of Electrical and Electronics Engineers, 2008.

[101] Insititue of Electrical and Electronics Engineers. SystemVerilog – unified hardware design, specification, and verification language. Technical Report 1800–2012, Insititue of Electrical and Electronics Engineers, 2012.

[102] Intel. Cilk Plus language extension specification. Technical report, Intel, 2011.

[103] Intel. Parallel Amplifier. http://software.intel.com/en-us/intel-vtune-amplifier-xe/, 2013.

[104] Ecma International. C# language specification 4th edition. Technical Report ECMA-334, Ecma International, 2006.

[105] Ecma International. Eiffel: Analysis, design and programming language 2nd edition. Technical Report ECMA-367, Ecma International, 2006.

[106] International Organization for Standardization. Prolog – part 1: General core. Technical Report 13211-1:1995, International Organization for Standardization, 1995.

[107] International Organization for Standardization. Extended BNF. Technical Report ISO/IEC 14977:1996, International Organization for Standardization, 1996.

[108] International Organization for Standardization. Prolog – part 2: Modules. Technical Report 13211-2:2000, International Organization for Standardization, 2000.

[109] International Organization for Standardization. Unified Modeling Language (UML) Version 1.4.2. Technical Report ISO/IEC 19501:2005, International Organization for Standardization, 2005.

[110] International Organization for Standardization. Technical Report 13211-1:1995/Cor 1:2007, International Organization for Standardization, 2007.

[111] International Organization for Standardization. C. Technical Report ISO/IEC 9899:2011, International Organization for Standardization, 2011.

[112] International Organization for Standardization. C++. Technical Report ISO/IEC 14882:2011, International Organization for Standardization, 2011.

[113] International Organization for Standardization. Technical Report ISO/IEC 9899:2011/Cor 1:2012, International Organization for Standardization, 2012.

[114] International Organization for Standardization. Technical Report 13211-1:1995/Cor 2:2012, International Organization for Standardization, 2012.

[115] International Organization for Standardization. Ada. Technical Report ISO/IEC 8652:2012, International Organization for Standardization, 2012.

[116] Valérie Issarny. An exception handling mechanism for parallel object-oriented programming: towards reusable, robust distributed software. *Journal of Object-Oriented Programming*, 6(6):29–39, 1993.

[117] Douglas L. Jones. Decimation-in-time (DIT) radix-2 FFT. `http://cnx.org/content/m12016/1.7/`, 2013.

[118] Mackale Joyner, Bradford L. Chamberlain, and Steven J. Deitz. Iterators in Chapel. In *International Parallel and Distributed Processing Symposium*, 2006.

[119] Horatiu Jula and Nicu G. Fruja. An executable specification of C#. In *Abstract State Machines*, pages 275–288, 2005.

[120] Gilles Kahn. Natural semantics. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39, 1987.

[121] Setrag Khoshafian and George P. Copeland. Object identity. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 406–416, 1986.

[122] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[123] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir performance analysis tool-set. In *International Parallel Tools Workshop*, pages 139–155, 2008.

[124] David A. Kranz, Kirk L. Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: Early experience. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 54–63, 1993.

[125] Roy Krischer. *Advanced Concepts in Asynchronous Exception Handling*. PhD thesis, University of Waterloo, 2010.

[126] Marija Kulas and Christoph Beierle. Defining standard Prolog in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 36:158–174, 2000.

[127] Dawid Kurzyniec and Vaidy S. Sunderam. Semantic aspects of asynchronous RMI: the RMIX approach. In *International Parallel and Distributed Processing Symposium*, 2004.

[128] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[129] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[130] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[131] Guodong Li. *Formal Verification of Programs and Their Transformations*. PhD thesis, University of Utah, 2010.

[132] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 2013.

[133] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl. Argus reference manual. Technical report, Massachusetts Institute of Technology, 1995.

[134] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Notices*, 23(7):260–267, 1988.

[135] Barbara Liskov and Stephen Zilles. Programming with abstract data types. *ACM SIGPLAN Notices*, 9(4):50–59, 1974.

[136] Andreas Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler*. PhD thesis, Karlsruher Institute of Technology, 2012.

[137] Daniel Lorenz, David Böhme, Bernd Mohr, Alexandre Strube, and Zoltán Szebenyi. Extending Scalasca's analysis features. In *International Parallel Tools Workshop*, pages 115–126, 2012.

[138] LORIA. SmartEiffel. `http://smarteiffel.loria.fr/`, 2013.

[139] Allen D. Malony, Sameer Shende, Wyatt Spear, Chee Wai Lee, and Scott Biersdorff. Advances in the tau performance system. In *International Parallel Tools Workshop*, pages 119–130, 2011.

[140] Simon Marlow, Simon Peyton-Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. *SIGPLAN Notices*, 36(5):274–285, 2001.

[141] Friedemann Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[142] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[143] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 179–188, 2010.

[144] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[145] José Meseguer. Membership algebra as a logical framework for equational specification. In *Workshop on Algebraic Development Techniques*, pages 18–61, 1997.

[146] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical report, Message Passing Interface Forum, 2012.

[147] Bertrand Meyer. A three-level approach to data structure description, and notational framework. In *Workshop on Data Abstraction, Databases and Conceptual Modelling*, pages 164–166, 1981.

[148] Bertrand Meyer. Sequential and concurrent object-oriented programming. In *Technology of Object-Oriented Languages and Systems*, pages 17–28, 1990.

[149] Bertrand Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.

[150] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd edition, 1997.

[151] Bertrand Meyer, Alexander Kogtenkov, and Anton Akhi. Processors and their collection. In *International Conference on Multicore Software Engineering, Performance, and Tools*, pages 1–15, 2012.

[152] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[153] Barton P. Miller, Morgan Clark, Jeffrey K. Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: the second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, 1990.

[154] Robert Miller and Anand R. Tripathi. The guardian model and primitives for exception handling in distributed systems. *IEEE Transactions on Software Engineering*, 30(12):1008–1022, 2004.

[155] David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space*. CRC Press, 2nd edition, 2010.

[156] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[157] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Software and Systems Modeling*, 6(1):83–110, 2007.

[158] Bernd Mohr and Felix Wolf. KOJAK - a tool set for automatic performance analysis of parallel programs. In *European Conference on Parallel Processing*, pages 1301–1304, 2003.

[159] Mohammad Reza Mousavi, Marjan Sirjani, and Farhad Arbab. Specification and verification of component connectors. Technical Report CSR-04-15, Eindhoven University of Technology, 2004.

[160] Mohammad Reza Mousavi, Marjan Sirjani, and Farhad Arbab. Formal semantics and analysis of component connectors in Reo. *Electronic Notes in Theoretical Computer Science*, 154(1):83–99, 2006.

[161] Sebastian Nanz, Faraz Torshizi, Michela Pedroni, and Bertrand Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. *Information and Software Technology*, 55(7):1304–1315, 2013.

[162] Stas Negara, Rajesh K. Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 81–90, 2011.

[163] Greg Nelson. *Systems Programming With Modula-3*. Prentice Hall, 1991.

[164] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zurich, 2007.

[165] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[166] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[167] Michael Norrish. A formal semantics for C++. Technical report, NICTA, 2007.

[168] Martin Odersky. The Scala language specification version 2.8. Technical report, ETH Lausanne, 2013.

[169] University of Texas at Austin. Orc reference manual v2.1.0. Technical report, University of Texas at Austin, 2013.

[170] Ronald A. Olsson and Aaron W. Keen. *The JR Programming Language*. Kluwer Academic Publishers, 2004.

[171] Peter Csaba Ölveczky and José Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1–2):161–196, 2007.

[172] OpenMP Architecture Review Board. OpenMP application program interface. Technical report, OpenMP Architecture Review Board, 2013.

[173] Jonathan S. Ostroff, Faraz A. Torshizi, Hai F. Huang, and Bernd Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing*, 21(4):319–346, 2008.

[174] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *International Conference on Theorem Proving in Higher Order Logics*, pages 391–407, 2009.

[175] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752, 1992.

[176] Robert L. Palmer. *Formal Analysis for MPI-Based High Performance Computing Software*. PhD thesis, University of Utah, 2007.

[177] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Technical University Darmstadt, 1962.

[178] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, pages 202–206, 1999.

[179] Gordon D. Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[180] Amir Pnueli. The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[181] Boris V. Protopopov and Anthony Skjellum. Shared-memory communication approaches for an MPI message-passing library. *Concurrency and Computation: Practice and Experience*, 12(9):799–820, 2000.

[182] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 427–436, 2009.

[183] Rajeev R. Raje, Joseph I. Williams, and Michael Boyles. Asynchronous remote method invocation (ARMI) mechanism for Java. *Concurrency and Computation: Practice and Experience*, 9(11):1207–1211, 1997.

[184] Ganesh Ramanathan, Benjamin Morandi, Scott West, Sebastian Nanz, and Bertrand Meyer. Deriving concurrent control software from behavioral specifications. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1994–1999, 2010.

[185] Brian Randell, Alexander Romanovsky, Robert J. Stroud, Jie Xu, and Avelino F. Zorzo. Coordinated atomic actions: from concept to implementation. Technical Report 595, University of Newcastle upon Tyne, 1997.

[186] Matti Rintala. Handling multiple concurrent exceptions in C++ using futures. In *Advances in Exception Handling Techniques*, pages 62–80, 2006.

[187] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[188] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. Technical report, IBM, 2011.

[189] Andreas Schedler. Conceptualizing accountability. In Andreas Schedler, Larry Diamond, and Marc F. Plattner, editors, *The Self-Restraining State: Power and Accountability in New Democracies*, pages 13–28. Lynne Rienner, 1999.

[190] Mischael Schill, Sebastian Nanz, and Bertrand Meyer. Handling parallelism in a concurrency model. In *International Conference on Multicore Software Engineering, Performance, and Tools*, pages 37–48, 2013.

[191] Joachim Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, Ulm University, 2002.

[192] Heinz W. Schmidt and Jian Chen. Reasoning about concurrent objects. In *Asia-Pacific Software Engineering Conference*, pages 86–95, 1995.

[193] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. volume 20, pages 71–122, 2010.

[194] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Z. Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, 2010.

[195] SGS-Thomson. occam 2.1 reference manual. Technical report, SGS-Thomson, 1995.

[196] Sameer Shende and Allen D. Malony. The Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[197] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32, 2008.

[198] Robert Snelick. S-Check: a tool for tuning parallel programs. In *International Parallel Processing Symposium*, pages 107–112, 1997.

[199] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. *Revised Report [6] on the Algorithmic Language Scheme*. Cambridge University Press, 2010.

[200] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.

[201] Hung-Hsun Su, Max Billingsley III, and Alan D. George. Parallel performance wizard: a performance system for the analysis of partitioned global-address-space applications. *International Journal of High Performance Computing Applications*, 24(4):485–510, 2010.

[202] Nathan R. Tallent and John M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–240, 2009.

[203] Prasanna Thati, Koushik Sen, and Narciso Martí-Oliet. An executable specification of asynchronous Pi-calculus semantics and may testing in Maude 2.0. *Electronic Notes in Theoretical Computer Science*, 71:261–281, 2004.

[204] Emina Torlak, Mandana Vaziri, and Julian Dolby. MemSAT: Checking axiomatic specifications of memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 341–350, 2010.

[205] Faraz Torshizi, Jonathan S. Ostroff, Richard F. Paige, and Marsha Chechik. The SCOOP concurrency model in Java-like languages. In *Communicating Process Architectures Conference*, pages 155–178, 2009.

[206] Anand R. Tripathi and Robert Miller. Exception handling in agent-oriented systems. In *Advances in Exception Handling Techniques*, pages 128–146, 2000.

[207] Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *International Conference on Computer Aided Verification*, pages 66–79, 2008.

[208] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[209] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006.

[210] Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *Asian Symposium on Programming Languages and Systems*, pages 194–209, 2009.

[211] Daniel Wasserrab. *From Formal Semantics to Verified Slicing: A Modular Framework with Applications in Language Based Security*. PhD thesis, Karlsruher Institute of Technology, 2010.

[212] Ian Wehrman, David Kitchin, William R. Cook, and Jayadev Misra. Properties of the timed operational and denotational semantics of Orc. Technical Report TR-07-65, University of Texas at Austin, 2007.

[213] Ian Wehrman, David Kitchin, William R. Cook, and Jayadev Misra. A timed semantics of Orc. *Theoretical Computer Science*, 402(2–3):234–248, 2008.

[214] Fangzhou Wei and Ali E. Yilmaz. A hybrid message passing/shared memory parallelization of the adaptive integral method for multi-core clusters. *Parallel Computing*, 37(6):279–301, 2011.

[215] Scott West, Sebastian Nanz, and Bertrand Meyer. A modular scheme for deadlock prevention in an object-oriented programming model. In *International Conference on Formal Engineering Methods*, pages 597–612, 2010.

[216] Jeannette M. Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietzke, Richard Lerner, and Su-Yuen Ling. The Avalon/C++ programming language (version 0). Technical report, Carnegie Mellon University, 1988.

[217] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10–11):421–439, 2003.

[218] H'sien J. Wong and Alistair P. Rendell. Integrating software distributed shared memory and message passing programming. In *IEEE International Conference on Cluster Computing*, pages 1–10, 2009.

[219] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecilia M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *International Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.

[220] Cui-Qing Yang and Barton P. Miller. Critical path analysis for the execution of parallel and distributed programs. In *International Conference on Distributed Computing Systems*, pages 366–373, 1988.

[221] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. Specifying Java thread semantics using a uniform memory model. In *Joint ACM-ISCOPE Conference on Java Grande*, pages 192–201, 2002.

[222] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *International Parallel and Distributed Processing Symposium*, 2004.

[223] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA specifications. In *Conference on Correct Hardware Design and Verification Methods*, pages 54–66, 1999.

List of figures

List of tables

A SCOOP_14 reference

A.1 *ID*

The universal stateful query *new_id* : *ID* returns a fresh identifier of type *ID*.

A.2 *NAME*

NAME represents names.

A.3 *PROGRAM*

A.3.1 Queries

$classes\colon PROGRAM \rightarrow SET\langle CLASS\_TYPE\rangle$
$settings\colon PROGRAM \rightarrow SETTINGS$

A.3.2 Constructors

$make\colon SET\langle CLASS\_TYPE\rangle \rightarrow SETTINGS \rightarrow PROGRAM$
  axioms
    $make(\overline{c}, st).classes = \overline{c}$
    $make(\overline{c}, st).settings = st$

A.4 *SETTINGS*

A.4.1 Queries

$root\_class\colon SETTINGS \rightarrow CLASS\_TYPE$
$root\_procedure\colon SETTINGS \rightarrow FEATURE$
$safe\_mode\colon SETTINGS \rightarrow BOOLEAN$

A.4.2 Constructors

$make\colon CLASS\_TYPE \rightarrow FEATURE \rightarrow BOOLEAN \rightarrow SETTINGS$

axioms

$make(c, f, sm).root\_class = c$
$make(c, f, sm).root\_procedure = f$
$make(c, f, sm).safe\_mode = sm$

A.5 *CLASS_TYPE*

A.5.1 Queries

$id\colon CLASS\_TYPE \rightarrow ID$
$name\colon CLASS\_TYPE \rightarrow NAME$
$is\_ref\colon CLASS\_TYPE \rightarrow BOOLEAN$
$is\_exp\colon CLASS\_TYPE \rightarrow BOOLEAN$
$attributes\colon CLASS\_TYPE \rightarrow SET\langle FEATURE\rangle$
$functions\colon CLASS\_TYPE \rightarrow SET\langle FEATURE\rangle$
$procedures\colon CLASS\_TYPE \rightarrow SET\langle FEATURE\rangle$
$has\_inv\colon CLASS\_TYPE \rightarrow BOOLEAN$
$inv\colon CLASS\_TYPE \rightarrow EXPRESSION$
$program\colon CLASS\_TYPE \rightarrow PROGRAM$
$feature\_by\_name\colon CLASS\_TYPE \rightarrow NAME \nrightarrow FEATURE$

$c.feature\_by\_name(n)$ **require**

$\exists f \in (c.attributes \cup c.functions \cup c.procedures)\colon f.name = n$

axioms

$c.feature\_by\_name(n) = f$

where

$f \in (c.attributes \cup c.functions \cup c.procedures) \wedge f.name = n$

A.5.2 Constructors

make: *ID* → *NAME* → *BOOLEAN* → *BOOLEAN* → *SET*⟨*FEATURE*⟩ →
SET⟨*FEATURE*⟩ → *SET*⟨*FEATURE*⟩ → *BOOLEAN* → *EXPRESSION*
→ *PROGRAM* → *CLASS_TYPE*

axioms

$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).id = id$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).name = n$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).is\_ref = ir$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).is\_exp = ie$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).attributes = \overline{f_a}$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).functions = \overline{f_f}$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).procedures = \overline{f_p}$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).has\_inv = hinv$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).inv = inv$
$make(id, n, ir, ie, \overline{f_a}, \overline{f_f}, \overline{f_p}, hinv, inv, prg).program = prg$

A.5.3 Instances

One instance of *CLASS_TYPE* is *boolean*, the class type of boolean objects. The class type *boolean* is expanded. It declares the attribute *item*, storing a boolean value as an instance of *BOOLEAN*.

A.6 *FEATURE*

A.6.1 Queries

id: *FEATURE* → *ID*
name: *FEATURE* → *NAME*
formals: *FEATURE* → *TUPLE*⟨*ENTITY*, . . . , *ENTITY*⟩
is_once: *FEATURE* → *BOOLEAN*
has_pre: *FEATURE* → *BOOLEAN*
pre: *FEATURE* → *EXPRESSION*
has_post: *FEATURE* → *BOOLEAN*
post: *FEATURE* → *EXPRESSION*
locals: *FEATURE* → *TUPLE*⟨*ENTITY*, . . . , *ENTITY*⟩
body: *FEATURE* → *STATEMENT_SEQUENCE*
rescue_clause: *FEATURE* → *STATEMENT_SEQUENCE*
is_exported: *FEATURE* → *BOOLEAN*
class_type: *FEATURE* → *CLASS_TYPE*

A.6.2 Constructors

$make\colon ID \rightarrow NAME \rightarrow TUPLE\langle ENTITY, \ldots, ENTITY\rangle \rightarrow BOOLEAN \rightarrow BOOLEAN$
$\quad\rightarrow EXPRESSION \rightarrow BOOLEAN \rightarrow EXPRESSION$
$\quad\rightarrow TUPLE\langle ENTITY, \ldots, ENTITY\rangle \rightarrow STATEMENT\_SEQUENCE$
$\quad\rightarrow STATEMENT\_SEQUENCE \rightarrow BOOLEAN \rightarrow CLASS\_TYPE \rightarrow FEATURE$

axioms

$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).id = id$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).name = n$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).formals = \overline{b_f}$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).is\_once = io$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).has\_pre = hpre$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).pre = pre$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).has\_post = hpost$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).post = post$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).locals = \overline{b_l}$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).body = body$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).rescue\_clause = resc$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).is\_exported = ie$
$make(id, n, \overline{b_f}, io, hpre, pre, hpost, post, \overline{b_l}, body, resc, ie, c).class\_type = c$

A.6.3 Subtypes

FEATURE has four subtypes: *ROUTINE* for routines, *FUNCTION* for functions, *PROCEDURE* for procedures, and *ATTRIBUTE* for attributes.

A.6.4 Instances

Three instances of *FEATURE* are *raise* to raise an exception as well as *set_passive* and *set_active* to set a processor of a given expression passive or active.

A.7 *ENTITY*

A.7.1 Queries

$name\colon ENTITY \rightarrow NAME$

A.7.2 Constructors

$make\colon NAME \to ENTITY$

axioms

$make(n).name = n$

A.7.3 Subtypes

ENTITY has one subtype *ATTRIBUTE* representing attributes.

A.7.4 Instances

Two instances of *ENTITY* are *current*, i.e., the entity for the current object, and *result*, i.e., the entity for the result of a function.

A.8 *STATEMENT* and *STATEMENT_SEQUENCE*

```
STATEMENT_SEQUENCE ≜ [STATEMENT {; STATEMENT}] ;
STATEMENT ≜ INSTRUCTION | OPERATION ;
```

A.9 *INSTRUCTION*

```
INSTRUCTION ≜ COMMAND | CREATION | IF | LOOP | ASSIGNMENT ;
COMMAND ≜
    EXPRESSION . FEATURE TUPLE⟨EXPRESSION, ..., EXPRESSION⟩ ;
CREATION ≜
    create ENTITY . FEATURE TUPLE⟨EXPRESSION, ..., EXPRESSION⟩ ;
IF ≜
    if EXPRESSION then
        STATEMENT_SEQUENCE
    [else
        STATEMENT_SEQUENCE]
    end ;
LOOP ≜ until EXPRESSION loop STATEMENT_SEQUENCE end ;
ASSIGNMENT ≜ ENTITY := EXPRESSION ;
RETRY ≜ retry ;
```

A.10 *OPERATION*

```
OPERATION ≜ ISSUE | RESULT | NOTIFY | WAIT | EVAL | LOCK |
    UNLOCK_RQ | POP_OBTAINED_LOCKS | WRITE | READ | PROVIDED |
    NOP | CALL | APPLY | CHECK_PRE_AND_LOCK | EXECUTE_BODY |
    EXECUTE_RESCUE_CLAUSE | SET_NOT_FRESH | CHECK_POST_AND_INV |
    RETURN | PURGE | FAILURE_ANCHOR | FAIL ;
ISSUE ≜ issue TUPLE⟨PROC, STATEMENT_SEQUENCE⟩ ;
RESULT ≜ result TUPLE⟨CHANNEL, REF⟩ ;
NOTIFY ≜ notify TUPLE⟨CHANNEL, BOOLEAN⟩ ;
WAIT ≜ wait TUPLE⟨CHANNEL, PROC⟩ ;
EVAL ≜ eval TUPLE⟨CHANNEL, EXPRESSION⟩ ;
LOCK ≜ lock TUPLE⟨SET⟨PROC⟩⟩ ;
UNLOCK_RQ ≜ unlock_rq ;
POP_OBTAINED_LOCKS ≜ pop_obtained_locks ;
WRITE ≜ write TUPLE⟨ENTITY, (REF | PROC), BOOLEAN⟩ ;
READ ≜ read TUPLE⟨ENTITY, CHANNEL⟩ ;
PROVIDED ≜
    provided (REF | BOOLEAN) then
        STATEMENT_SEQUENCE
    else
        STATEMENT_SEQUENCE
    end ;
NOP ≜ nop ;
CALL ≜ call TUPLE⟨REF, FEATURE, TUPLE⟨REF, . . . , REF⟩⟩ |
    call TUPLE⟨CHANNEL, REF, FEATURE, TUPLE⟨REF, . . . , REF⟩⟩ ;
APPLY ≜ apply TUPLE⟨CHANNEL, REF, FEATURE, TUPLE⟨REF, . . . , REF⟩,
     PROC, TUPLE⟨SET⟨PROC⟩, SET⟨PROC⟩⟩⟩ ;
CHECK_PRE_AND_LOCK ≜
    check_pre_and_lock TUPLE⟨CHANNEL, FEATURE, PROC,
    TUPLE⟨SET⟨PROC⟩, SET⟨PROC⟩⟩, SET⟨PROC⟩, PROC⟩ ;
EXECUTE_BODY ≜ execute_body TUPLE⟨FEATURE, SET⟨PROC⟩⟩ ;
EXECUTE_RESCUE_CLAUSE ≜ execute_rescue_clause TUPLE⟨FEATURE⟩ ;
SET_NOT_FRESH ≜ set_not_fresh TUPLE⟨FEATURE⟩ ;
CHECK_POST_AND_INV ≜ check_post_and_inv TUPLE⟨FEATURE⟩ ;
RETURN ≜
    return TUPLE⟨CHANNEL, FEATURE, PROC,
        TUPLE⟨SET⟨PROC⟩, SET⟨PROC⟩⟩, PROC,
         BOOLEAN, BOOLEAN, BOOLEAN⟩ |
    return TUPLE⟨CHANNEL, FEATURE, REF, PROC,
        TUPLE⟨SET⟨PROC⟩, SET⟨PROC⟩⟩, PROC,
         BOOLEAN, BOOLEAN, BOOLEAN⟩ ;
```

```
PURGE ≜ purge ;
FAILURE_ANCHOR ≜
  failure_anchor BOOLEAN
    final STATEMENT_SEQUENCE
    normal STATEMENT_SEQUENCE
    rescue BOOLEAN STATEMENT_SEQUENCE
    retry STATEMENT_SEQUENCE
    failure STATEMENT_SEQUENCE
  end ;
FAIL ≜ fail |
  fail
    rescue BOOLEAN STATEMENT_SEQUENCE
    retry STATEMENT_SEQUENCE
    failure STATEMENT_SEQUENCE
  end ;
```

A.11 *EXPRESSION*

```
EXPRESSION ≜ ENTITY | LITERAL | QUERY ;
```

A.12 *LITERAL*

```
LITERAL = BOOL_LITERAL | INT_LITERAL | CHAR_LITERAL | VOID_LITERAL ;
BOOL_LITERAL ≜ True | False ;
INT_LITERAL ≜ [-](0 | ... | 9){0 | ... | 9} ;
CHAR_LITERAL ≜ ' (a | ... | z | A | ... | Z | 0 | ... | 9) ' ;
VOID_LITERAL ≜ Void ;
```

The universal function *obj* : *OBJ* takes a literal and returns a new object matching the literal in both type and value.

A.13 *QUERY*

```
QUERY ≜
  EXPRESSION . FEATURE TUPLE⟨EXPRESSION, ..., EXPRESSION⟩ ;
```

A.14 Types

A type t is a triple (d, p, c): d is the detachable tag, p is the processor tag, and c is the class type of type *CLASS_TYPE*. The detachable tag d specifies detachability:

- If $d = !$, then the typed element is statically guaranteed to hold a value, i.e., to be non-void.
- If $d = ?$, then the typed element can be void.

The processor tag p specifies locality:

- If $p = \top$, then the typed element points to a potentially different processor.
- If $p =< x.handler >$ where x is of type *ENTITY*, then the typed element points to the processor of the object in x.
- If $p =< x >$ where x is of type *ENTITY*, then the typed element points to the processor in x.
- If $p = \bullet$, then the typed element points to the current processor.

The typing environment Γ, as formalized by Nienaltowski [164], contains the class hierarchy of the program along with all the type definitions for all features and entities. The predicate $\Gamma \vdash e : t$ denotes that expression e is of type t. The function $type\_of(\Gamma, e)$ returns the type of expression e in Γ. The predicate $controlled(\Gamma, t)$ denotes that expression e of type t is controlled. The function $controlling\_entity(\Gamma, e)$ returns the controlling entity for an expression e as an instance of *ENTITY*.

A.15 *REF*

A.15.1 Queries

$id\colon REF \to ID$

A.15.2 Constructors

$make\colon REF$

axioms

$make.id = new\_id$

A.15.3 Instances

One instance of *REF* is *void*, the void reference.

A.16 *OBJ*

A.16.1 Queries

$id\colon OBJ \to ID$

$class\_type\colon OBJ \to CLASS\_TYPE$

$att\_val\colon OBJ \to ATTRIBUTE \nrightarrow REF \cup PROC$

$o.att\_val(f)$ **require**

$o.class\_type.attributes.has(f)$

$copy\colon OBJ \to OBJ$

axioms

$o.copy = make(o.class\_type)$

$.set\_att\_val(a_1, o.att\_val(a_1))$

$. \; ...$

$.set\_att\_val(a_n, o.att\_val(a_n))$

where

$\{a_1, \ldots, a_n\} \overset{def}{=} o.class\_type.attributes$

A.16.2 Commands

$set\_att\_val \colon OBJ \to ATTRIBUTE \nrightarrow REF \cup PROC \to OBJ$
  $o.set\_att\_val(f, v)$ **require**
    $o.class\_type.attributes.has(f)$
  axioms
    $o.set\_att\_val(f, v).att\_val(f) = v$

A.16.3 Constructors

$make \colon CLASS\_TYPE \to OBJ$
  axioms
    $make(c).id = new\_id$
    $make(c).class\_type = c$
    $\forall i \in \{1, \ldots, n\} \colon make(c).att\_val(a_i) = void$
     where
      $\{a_1, \ldots, a_n\} \overset{def}{=} c.attributes$

A.17 *HEAP*

A.17.1 Queries

$objs \colon HEAP \to SET\langle OBJ \rangle$
$refs \colon HEAP \to SET\langle REF \rangle$
$ref\_obj \colon HEAP \to REF \nrightarrow OBJ$
  $h.ref\_obj(r)$ **require**
    $h.refs.has(r)$
$last\_added\_obj \colon HEAP \to REF$
  $h.last\_added\_obj$ **require**
    $\neg h.refs.empty$
$ref \colon HEAP \to OBJ \nrightarrow REF$
  $h.ref(o)$ **require**
    $h.objs.has(o)$
  axioms
    $h.ref\_obj(h.ref(o)) = o$
$fresh \colon HEAP \to PROC \to ID \nrightarrow BOOLEAN$

$stable\colon HEAP \rightarrow PROC \rightarrow ID \nrightarrow BOOLEAN$
    $h.stable(p, i)$ **require**
        $\neg h.fresh(p, i)$
$stabilizer\colon HEAP \rightarrow PROC \rightarrow ID \nrightarrow PROC$
    $h.stabilizer(p, i)$ **require**
        $\neg h.fresh(p, i)$
$has\_failed\colon HEAP \rightarrow PROC \rightarrow ID \nrightarrow BOOLEAN$
    $h.has\_failed(p, i)$ **require**
        $\neg h.fresh(p, i)$
$once\_result\colon HEAP \rightarrow PROC \rightarrow ID \nrightarrow REF$
    $h.once\_result(p, i)$ **require**
        $\neg h.fresh(p, i)$

A.17.2 Commands

$add\_obj\colon HEAP \rightarrow OBJ \nrightarrow HEAP$
    $h.add\_obj(o)$ **require**
        $\forall u \in h.objs\colon u.id \neq o.id$
        $\forall a \in o.class\_type.attributes\colon$
            $o.att\_val(a) \in REF \Rightarrow (o.att\_val(a) = void \vee h.refs.has(o.att\_val(a)))$
    axioms
        $h.add\_obj(o).objs = h.objs \cup \{o\}$
        $h.add\_obj(o).refs = h.refs \cup \{r\}$
        $h.add\_obj(o).ref\_obj(r) = o$
        $h.add\_obj(o).last\_added\_obj = r$
          where
            $r \overset{def}{=}$ **new** $REF.make$
$update\_obj\colon HEAP \rightarrow REF \nrightarrow OBJ \nrightarrow HEAP$
    $h.update\_obj(r, o)$ **require**
        $h.refs.has(r)$
        $o.id = h.ref\_obj(r).id$
        $\forall a \in o.class\_type.attributes\colon$
            $o.att\_val(a) \in REF \Rightarrow (o.att\_val(a) = void \vee h.refs.has(o.att\_val(a)))$
    axioms
        $h.update\_obj(r, o).objs.has(o)$
        $o \neq h.ref\_obj(r) \Rightarrow \neg h.update\_obj(r, o).objs.has(h.ref\_obj(r))$
        $h.update\_obj(r, o).ref\_obj(r) = o$

$set\_once\_func\_not\_fresh\colon HEAP \to PROC \to FEATURE \nrightarrow BOOLEAN \to BOOLEAN$
$\quad \to REF \nrightarrow HEAP$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r)$ **require**

$f \in FUNCTION \wedge f.is\_once$

$r \neq void \Rightarrow h.refs.has(r)$

axioms

$(\exists d, c\colon \Gamma \vdash f : (d, \bullet, c)) \Rightarrow$

$\neg h.set\_once\_func\_not\_fresh(p, f, st, hf, r).fresh(p, f.id) \wedge$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).stable(p, f.id) = st \wedge$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).stabilizer(p, f.id) = p \wedge$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).has\_failed(p, f.id) = hf \wedge$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).once\_result(p, f.id) = \begin{cases} r & \text{if } \neg hf \\ void & \text{if } hf \end{cases}$

$(\exists d, c\colon \Gamma \vdash f : (d, p, c) \wedge p \neq \bullet) \Rightarrow \forall q \in PROC\colon$

$\neg h.set\_once\_func\_not\_fresh(p, f, st, hf, r).fresh(q, f.id) \wedge$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).stable(q, f.id) = st \wedge$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).stabilizer(q, f.id) = p \wedge$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).has\_failed(q, f.id) = hf \wedge$

$h.set\_once\_func\_not\_fresh(p, f, st, hf, r).once\_result(q, f.id) = \begin{cases} r & \text{if } \neg hf \\ void & \text{if } hf \end{cases}$

$set\_once\_proc\_not\_fresh\colon HEAP \to PROC \to FEATURE \nrightarrow BOOLEAN \to BOOLEAN$
$\quad \to HEAP$

$h.set\_once\_proc\_not\_fresh(p, f, st, hf)$ **require**

$f \in PROCEDURE \wedge f.is\_once$

axioms

$\neg h.set\_once\_proc\_not\_fresh(p, f, st, hf).fresh(p, f.id)$

$h.set\_once\_proc\_not\_fresh(p, f, st, hf).stable(p, f.id) = st$

$h.set\_once\_proc\_not\_fresh(p, f, st, hf).stabilizer(p, f.id) = p$

$h.set\_once\_proc\_not\_fresh(p, f, st, hf).has\_failed(p, f.id) = hf$

$set\_once\_rout\_fresh\colon HEAP \to PROC \to FEATURE \nrightarrow HEAP$

$h.set\_once\_rout\_fresh(p, f)$ **require**

$f.is\_once$

axioms

$(f \in FUNCTION \wedge \exists d, c\colon \Gamma \vdash f : (d, \bullet, c)) \Rightarrow$

$h.set\_once\_rout\_fresh(p, f).fresh(p, f.id)$

$(f \in FUNCTION \wedge \exists d, c\colon \Gamma \vdash f : (d, p, c) \wedge p \neq \bullet) \Rightarrow \forall q \in PROC\colon$

$h.set\_once\_rout\_fresh(p, f).fresh(q, f.id)$

$(f \in PROCEDURE) \Rightarrow$

$h.set\_once\_rout\_fresh(p, f).fresh(p, f.id)$

A.17.3 Constructors

$make\colon HEAP$

  axioms

    $make.objs.empty$

    $make.refs.empty$

    $\forall p \in PROC, f \in FEATURE\colon\ f.is\_once \Rightarrow make.fresh(p, f.id)$

A.18 *PROC*

A.18.1 Queries

$id\colon PROC \to ID$

A.18.2 Constructors

$make\colon PROC$

  axioms

    $make.id = new\_id$

A.19 *REGIONS*

A.19.1 Queries

$procs\colon REGIONS \to SET\langle PROC\rangle$

$handled\_objs\colon REGIONS \to PROC \nrightarrow SET\langle REF\rangle$

  $k.handled\_objs(p)$ **require**

    $k.procs.has(p)$

$last\_added\_proc\colon REGIONS \nrightarrow PROC$

  $k.last\_added\_proc$ **require**

    $\neg k.procs.empty$

$handler\colon REGIONS \to REF \nrightarrow PROC$

  $k.handler(r)$ **require**

    $\exists p \in k.procs\colon\ k.handled\_objs(p).has(r)$

  axioms

    $k.handled\_objs(k.handler(o)).has(r)$

$rq\_locked\colon REGIONS \to PROC \nrightarrow BOOLEAN$
  $k.rq\_locked(p)$ **require**
    $k.procs.has(p)$
$cs\_locked\colon REGIONS \to PROC \nrightarrow BOOLEAN$
  $k.cs\_locked(p)$ **require**
    $k.procs.has(p)$
$obtained\_rq\_locks\colon REGIONS \to PROC \nrightarrow STACK\langle SET\langle PROC\rangle\rangle$
  $k.obtained\_rq\_locks(p)$ **require**
    $k.procs.has(p)$
$obtained\_cs\_lock\colon REGIONS \to PROC \nrightarrow PROC$
  $k.obtained\_cs\_lock(p)$ **require**
    $k.procs.has(p)$
$retrieved\_rq\_locks\colon REGIONS \to PROC \nrightarrow STACK\langle SET\langle PROC\rangle\rangle$
  $k.retrieved\_rq\_locks(p)$ **require**
    $k.procs.has(p)$
$retrieved\_cs\_locks\colon REGIONS \to PROC \nrightarrow STACK\langle SET\langle PROC\rangle\rangle$
  $k.retrieved\_cs\_locks(p)$ **require**
    $k.procs.has(p)$
$passed\colon REGIONS \to PROC \nrightarrow BOOLEAN$
  $k.passed(p)$ **require**
    $k.procs.has(p)$
$has\_failed\colon REGIONS \to PROC \nrightarrow BOOLEAN$
  $k.has\_failed(p)$ **require**
    $k.procs.has(p)$
$passive\colon REGIONS \to PROC \nrightarrow BOOLEAN$
  $k.passive(p)$ **require**
    $k.procs.has(p)$
$executes\_for\colon REGIONS \to PROC \nrightarrow PROC$
  $k.executes\_for(p)$ **require**
    $k.procs.has(p)$

A.19.2 Commands

$add\_obj\colon REGIONS \to PROC \nrightarrow REF \nrightarrow REGIONS$
  $k.add\_obj(p, r)$ **require**
    $k.procs.has(p)$
    $\forall q \in k.procs, x \in k.handled\_objs(q)\colon\ x.id \neq r.id$
  axioms
    $k.add\_obj(p, r).handled\_objs(p).has(r)$

```
$add\_proc\colon REGIONS \to PROC \nrightarrow REGIONS$
    $k.add\_proc(p)$ **require**
        $\neg k.procs.has(p)$
    **axioms**
        $k.add\_proc(p).procs.has(p)$
        $k.add\_proc(p).last\_added\_proc = p$
        $k.add\_proc(p).handled\_objs(p).empty$
        $\neg k.add\_proc(p).rq\_locked(p)$
        $k.add\_proc(p).cs\_locked(p)$
        $k.add\_proc(p).obtained\_rq\_locks(p).empty$
        $k.add\_proc(p).obtained\_cs\_lock(p) = p$
        $k.add\_proc(p).retrieved\_rq\_locks(p).empty$
        $k.add\_proc(p).retrieved\_cs\_locks(p).empty$
        $\neg k.add\_proc(p).passed(p)$
        $\neg k.add\_proc(p).has\_failed(p)$
        $\neg k.add\_proc(p).passive(p)$
        $k.add\_proc(p).executes\_for(p) = p$
$lock\_rqs\colon REGIONS \to PROC \nrightarrow SET\langle PROC\rangle \nrightarrow REGIONS$
    $k.lock\_rqs(p, \bar{l})$ **require**
        $k.procs.has(p)$
        $\forall x \in \bar{l}\colon k.procs.has(x)$
        $\forall x \in \bar{l}\colon \neg k.rq\_locked(x)$
    **axioms**
        $k.lock\_rqs(p, \bar{l}).obtained\_rq\_locks(p) = k.obtained\_rq\_locks(p).push(\bar{l})$
        $\forall x \in \bar{l}\colon k.lock\_rqs(p, \bar{l}).rq\_locked(x)$
$pop\_obtained\_rq\_locks\colon REGIONS \to PROC \nrightarrow REGIONS$
    $k.pop\_obtained\_rq\_locks(p)$ **require**
        $k.procs.has(p)$
        $\neg k.obtained\_rq\_locks(p).empty$
        $\neg k.passed(p)$
    **axioms**
        $k.pop\_obtained\_rq\_locks(p).obtained\_rq\_locks(p) = k.obtained\_rq\_locks(p).pop$
$unlock\_rq\colon REGIONS \to PROC \nrightarrow REGIONS$
    $k.unlock\_rq(p)$ **require**
        $k.procs.has(p)$
        $k.rq\_locked(p)$
        $\forall q \in k.procs\colon \neg k.obtained\_rq\_locks(q).flat.has(p)$
    **axioms**
        $\neg k.unlock\_rq(p).rq\_locked(p)$
```

$$\mathit{pass\_locks}\colon \mathit{REGIONS} \to \mathit{PROC} \nrightarrow \mathit{PROC} \nrightarrow \mathit{TUPLE}\langle \mathit{SET}\langle \mathit{PROC}\rangle, \mathit{SET}\langle \mathit{PROC}\rangle\rangle \nrightarrow \mathit{REGIONS}$$

$k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c}))$ **require**

$k.\mathit{procs.has}(p) \wedge k.\mathit{procs.has}(q)$

$\forall x \in \overline{l_r}\colon k.\mathit{obtained\_rq\_locks}(p).\mathit{flat.has}(x) \vee k.\mathit{retrieved\_rq\_locks}(p).\mathit{flat.has}(x)$

$\forall x \in \overline{l_c}\colon x = k.\mathit{obtained\_cs\_lock}(p) \vee k.\mathit{retrieved\_cs\_locks}(p).\mathit{flat.has}(x)$

$\neg\overline{l_r}.\mathit{empty} \vee \neg\overline{l_c}.\mathit{empty} \Rightarrow \neg k.\mathit{passed}(p)$

axioms

$\neg\overline{l_r}.\mathit{empty} \vee \neg\overline{l_c}.\mathit{empty} \Rightarrow k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\mathit{passed}(p)$

$k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\mathit{retrieved\_rq\_locks}(q) = k.\mathit{retrieved\_rq\_locks}(q).\mathit{push}(\overline{l_r})$

$k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})).\mathit{retrieved\_cs\_locks}(q) = k.\mathit{retrieved\_cs\_locks}(q).\mathit{push}(\overline{l_c})$

$$\begin{pmatrix} p \neq q \wedge \\ k.\mathit{passed}(q) \wedge \\ k.\mathit{obtained\_rq\_locks}(q).\mathit{flat} \subseteq \overline{l_r} \wedge \\ k.\mathit{retrieved\_rq\_locks}(q).\mathit{flat} \subseteq \overline{l_r} \wedge \\ k.\mathit{obtained\_cs\_lock}(q) \in \overline{l_c} \wedge \\ k.\mathit{retrieved\_cs\_locks}(q).\mathit{flat} \subseteq \overline{l_c} \end{pmatrix} \Rightarrow \begin{matrix} \neg k.\mathit{pass\_locks}(p, q, (\overline{l_r}, \overline{l_c})) \\ .\mathit{passed}(q) \end{matrix}$$

$$\mathit{revoke\_locks}\colon \mathit{REGIONS} \to \mathit{PROC} \nrightarrow \mathit{PROC} \nrightarrow \mathit{REGIONS}$$

$k.\mathit{revoke\_locks}(p, q)$ **require**

$k.\mathit{procs.has}(p) \wedge k.\mathit{procs.has}(q)$

$\neg k.\mathit{retrieved\_rq\_locks}(q).\mathit{empty} \wedge \neg k.\mathit{retrieved\_cs\_locks}(q).\mathit{empty}$

$k.\mathit{retrieved\_rq\_locks}(q).\mathit{top} \subseteq$
$\quad k.\mathit{obtained\_rq\_locks}(p).\mathit{flat} \cup k.\mathit{retrieved\_rq\_locks}(p).\mathit{flat}$

$k.\mathit{retrieved\_cs\_locks}(q).\mathit{top} \subseteq$
$\quad \{k.\mathit{obtained\_cs\_lock}(p)\} \cup k.\mathit{retrieved\_cs\_locks}(p).\mathit{flat}$

$k.\mathit{retrieved\_rq\_locks}(q).\mathit{top} \cup k.\mathit{retrieved\_cs\_locks}(q).\mathit{top} \neq \{\} \Rightarrow k.\mathit{passed}(p)$

$\neg k.\mathit{passed}(q)$

axioms

$k.\mathit{retrieved\_rq\_locks}(q).\mathit{top} \cup k.\mathit{retrieved\_cs\_locks}(q).\mathit{top} \neq \{\} \Rightarrow$
$\quad \neg k.\mathit{revoke\_locks}(p, q).\mathit{passed}(p)$

$k.\mathit{revoke\_locks}(p, q).\mathit{retrieved\_rq\_locks}(q) = k.\mathit{retrieved\_rq\_locks}(q).\mathit{pop}$

$k.\mathit{revoke\_locks}(p, q).\mathit{retrieved\_cs\_locks}(q) = k.\mathit{retrieved\_cs\_locks}(q).\mathit{pop}$

$$\begin{pmatrix} p \neq q \wedge \\ \begin{pmatrix} \exists x \in k.\mathit{retrieved\_rq\_locks}(p).\mathit{flat}\colon (\\ \quad k.\mathit{obtained\_rq\_locks}(q).\mathit{flat.has}(x) \vee \\ \quad k.\mathit{retrieved\_rq\_locks}(q).\mathit{pop.flat.has}(x) \\ ) \vee \\ \exists x \in k.\mathit{retrieved\_cs\_locks}(p).\mathit{flat}\colon (\\ \quad x = k.\mathit{obtained\_cs\_lock}(q) \vee \\ \quad k.\mathit{retrieved\_cs\_locks}(q).\mathit{pop.flat.has}(x) \\ ) \end{pmatrix} \end{pmatrix} \Rightarrow \begin{matrix} k.\mathit{revoke\_locks}(p, q) \\ .\mathit{passed}(q) \end{matrix}$$

set_failed: $REGIONS \to PROC \nrightarrow BOOLEAN \to REGIONS$
  $k.set\_failed(p, hf)$ **require**
    $k.procs.has(p)$
  axioms
    $k.set\_failed(p, hf).has\_failed(p) = hf$
$set\_passive$: $REGIONS \to PROC \nrightarrow BOOLEAN \to REGIONS$
  $k.set\_passive(p, ip)$ **require**
    $k.procs.has(p)$
    $ip \Rightarrow k.executes\_for(p) = p$
    $\neg ip \Rightarrow \neg\exists q \in k.procs\colon (q \neq p \land k.executes\_for(q) = p)$
  axioms
    $k.set\_passive(p, ip).passive(p) = ip$
$set\_executes\_for$: $REGIONS \to PROC \nrightarrow PROC \nrightarrow REGIONS$
  $k.set\_executes\_for(p, q)$ **require**
    $k.procs.has(p)$
    $k.procs.has(q)$
    $p \neq q \Rightarrow \neg k.passive(p) \land k.passive(q)$
  axioms
    $k.set\_executes\_for(p, q).executes\_for(p) = q$

A.19.3 Constructors

$make$: $REGIONS$
  axioms
    $make.procs.empty$

A.20 *ENV*

A.20.1 Queries

$names$: $ENV \to SET\langle NAME\rangle$
val: $ENV \to NAME \nrightarrow REF \cup PROC$
  $e.val(n)$ **require**
    $e.names.has(n)$

A.20.2 Commands

$update\colon ENV \to NAME \to REF \cup PROC \to ENV$

axioms

$e.update(n, v).names = e.names \cup \{n\}$

$e.update(n, v).val(n) = v$

A.20.3 Constructors

$make\colon ENV$

axioms

$make.names.empty$

A.21 *STORE*

A.21.1 Queries

$envs\colon STORE \to PROC \to STACK\langle ENV\rangle$

A.21.2 Commands

$push\_env\colon STORE \to PROC \to ENV \to STORE$

axioms

$s.push\_env(p, e).envs(p) = s.envs(p).push(e)$

$pop\_env\colon STORE \to PROC \nrightarrow STORE$

$s.pop\_env(p)$ **require**

$\neg s.envs(p).empty$

axioms

$s.pop\_env(p).envs(p) = s.envs(p).pop$

A.21.3 Constructors

$make\colon STORE$

axioms

$\forall p \in PROC\colon make.envs(p).empty$

A.22 *STATE*

A.22.1 Queries

$regions\colon STATE \to REGIONS$
$heap\colon STATE \to HEAP$
$store\colon STATE \to STORE$

A.22.2 Commands

$set\colon STATE \to REGIONS \nrightarrow HEAP \nrightarrow STORE \nrightarrow STATE$

```
  $\sigma.set(k, h, s)$ **require**
    $\forall p \in k.procs, r \in k.handled\_objs(p)\colon h.objs.has(h.ref\_obj(r))$
    $\forall o \in h.objs\colon \exists p \in k.procs\colon h.ref(o) \in k.handled\_objs(p)$
    $\forall p \in PROC, f \in FEATURE\colon \neg h.fresh(p, f.id) \Rightarrow k.procs.has(p)$
    $\forall o \in h.objs, a \in o.class\_type.attributes\colon$
      $o.att\_val(a) \in PROC \Rightarrow k.procs.has(o.att\_val(a))$
    $\forall p \in PROC, e \in s.envs(p)\colon \neg e.names.empty \Rightarrow k.procs.has(p)$
    $\forall p \in k.procs, e \in s.envs(p), x \in e.names\colon$
      $(e.val(x) \in REF \Rightarrow e.val(x) = void \vee h.refs.has(e.val(x)))\wedge$
      $(e.val(x) \in PROC \Rightarrow k.procs.has(e.val(x)))$
  **axioms**
    $\sigma.set(k, h, s).regions = k$
    $\sigma.set(k, h, s).heap = h$
    $\sigma.set(k, h, s).store = s$
```

A.22.3 Constructors

$make\colon STATE$

```
  **axioms**
    $make.regions =$ **new** $REGIONS.make$
    $make.heap =$ **new** $HEAP.make$
    $make.store =$ **new** $STORE.make$
```

A.23 *STATE* facade: mappings

A.23.1 Queries

$procs \colon STATE \to SET\langle PROC\rangle$
    axioms
        $\sigma.procs = \sigma.regions.procs$
$last\_added\_proc \colon STATE \nrightarrow PROC$
    $\sigma.last\_added\_proc$ **require**
        $\neg\sigma.regions.procs.empty$
    axioms
        $\sigma.last\_added\_proc = \sigma.regions.last\_added\_proc$
$handler \colon STATE \to REF \nrightarrow PROC$
    $\sigma.handler(r)$ **require**
        $\sigma.heap.refs.has(r)$
    axioms
        $\sigma.handler(r) = \sigma.regions.handler(r)$
$last\_added\_obj \colon STATE \nrightarrow REF$
    $\sigma.last\_added\_obj$ **require**
        $\neg\sigma.heap.refs.empty$
    axioms
        $\sigma.last\_added\_obj = \sigma.heap.last\_added\_obj$
$ref\_obj \colon STATE \to REF \nrightarrow OBJ$
    $\sigma.ref\_obj(r)$ **require**
        $\sigma.heap.refs.has(r)$
    axioms
        $\sigma.ref\_obj(r) = \sigma.heap.ref\_obj(r)$
$ref \colon STATE \to OBJ \nrightarrow REF$
    $\sigma.ref(o)$ **require**
        $\sigma.heap.objs.has(o)$
    axioms
        $\sigma.ref(o) = \sigma.heap.ref(o)$
$new\_proc \colon STATE \to PROC$
    axioms
        $\sigma.new\_proc =$ **new** $PROC.make$
$new\_obj \colon STATE \to CLASS\_TYPE \to OBJ$
    axioms
        $\sigma.new\_obj(c) =$ **new** $OBJ.make(c)$

A.23.2 Commands

$add\_proc\colon STATE \rightarrow PROC \nrightarrow STATE$
  $\sigma.add\_proc(p)$ **require**
    $\neg\sigma.regions.procs.has(p)$
  axioms
    $\sigma.add\_proc(p) = \sigma.set(\sigma.regions.add\_proc(p), \sigma.heap, \sigma.store)$

$add\_obj\colon STATE \rightarrow PROC \nrightarrow OBJ \nrightarrow STATE$
  $\sigma.add\_obj(p, o)$ **require**
    $\sigma.regions.procs.has(p)$
    $\forall u \in \sigma.heap.objs\colon u.id \neq o.id$
    $\forall a \in o.class\_type.attributes\colon$
      $(o.att\_val(a) \in REF \Rightarrow o.att\_val(a) = void \lor \sigma.heap.refs.has(o.att\_val(a)))\land$
      $(o.att\_val(a) \in PROC \Rightarrow \sigma.regions.procs.has(o.att\_val(a)))$
  axioms
    $\sigma.add\_obj(p, o) = \sigma.set(k, h, s)$
     where
     $h \overset{def}{=} \sigma.heap.add\_obj(o)$
     $k \overset{def}{=} \sigma.regions.add\_obj(p, h.last\_added\_obj)$
     $s \overset{def}{=} \sigma.store$

$update\_obj\colon STATE \rightarrow REF \nrightarrow OBJ \nrightarrow STATE$
  $\sigma.update\_obj(r, o)$ **require**
    $\sigma.heap.refs.has(r)$
    $o.id = \sigma.heap.ref\_obj(r).id$
    $\forall a \in o.class\_type.attributes\colon$
      $(o.att\_val(a) \in REF \Rightarrow o.att\_val(a) = void \lor \sigma.heap.refs.has(o.att\_val(a)))\land$
      $(o.att\_val(a) \in PROC \Rightarrow \sigma.regions.procs.has(o.att\_val(a)))$
  axioms
    $\sigma.update\_obj(r, o) = \sigma.set(\sigma.regions, \sigma.heap.update\_obj(r, o), \sigma.store)$

A.24 *STATE* facade: importing

A.24.1 Queries

$last\_imported\_obj\colon STATE \rightarrow REF$

A.24.2 Commands

$\mathit{import}\colon \mathit{STATE} \to \mathit{PROC} ⇸ \mathit{REF} ⇸ \mathit{STATE}$

$\quad \sigma.\mathit{import}(p, r)$ **require**

$\quad\quad \sigma.\mathit{regions.procs.has}(p)$

$\quad\quad \sigma.\mathit{heap.refs.has}(r)$

$\quad\quad \sigma.\mathit{heap.ref\_obj}(r).\mathit{class\_type.is\_exp}$

$\quad$ **axioms**

$\quad\quad \sigma.\mathit{import}(p, r) = \sigma'$

$\quad\quad \sigma.\mathit{import}(p, r).\mathit{last\_imported\_obj} = r'$

$\quad\quad\quad$ **where**

$\quad\quad\quad w \overset{def}{=}$ **new** $\mathit{MAP}\langle \mathit{REF}, \mathit{REF}\rangle.\mathit{make}$

$\quad\quad\quad (r', w', \sigma') \overset{def}{=} \mathit{import\_rec\_with\_map}(p, \sigma.\mathit{handler}(r), r, w, \sigma)$

$\mathit{import\_rec\_with\_map}(p, q, r, w, \sigma) \overset{def}{=} (r'', w'', \sigma'')$

where

$$(r', w', \sigma') \overset{def}{=} \mathit{import\_rec\_without\_map}(p, q, r, w, \sigma)$$

$$r'' \overset{def}{=} \begin{cases} w.\mathit{val}(r) & \text{if } w.\mathit{keys.has}(r) \\ r' & \text{if } \neg w.\mathit{keys.has}(r) \end{cases}$$

$$w'' \overset{def}{=} \begin{cases} w & \text{if } w.\mathit{keys.has}(r) \\ w' & \text{if } \neg w.\mathit{keys.has}(r) \end{cases}$$

$$\sigma'' \overset{def}{=} \begin{cases} \sigma & \text{if } w.\mathit{keys.has}(r) \\ \sigma' & \text{if } \neg w.\mathit{keys.has}(r) \end{cases}$$

$\mathit{import\_rec\_without\_map}(p, q, r, w, \sigma) \overset{def}{=} (r', w', \sigma')$

where

$$o'_0 \overset{def}{=} \sigma.\mathit{ref\_obj}(r).\mathit{copy}$$

$$\sigma'_0 \overset{def}{=} \sigma.\mathit{add\_obj}(p, o'_0)$$

$$w'_0 \overset{def}{=} w.\mathit{add}(r, \sigma'_0.\mathit{ref}(o'_0))$$

$$\{a_1, \ldots, a_n\} \overset{def}{=} \{a \mid o'_0.\mathit{att\_val}(a) \in \mathit{REF} \wedge o'_0.\mathit{att\_val}(a) \neq \mathit{void}\}$$

$$\forall i \in \{1, \ldots, n\}\colon (r'_i, w'_i, \sigma'_i) \overset{def}{=} \mathit{import\_rec\_with\_map}(p, q, o'_0.\mathit{att\_val}(a_i), w'_{i-1}, \sigma'_{i-1})$$

$$\forall i \in \{1, \ldots, n\}\colon o'_i \overset{def}{=} o'_{i-1}.\mathit{set\_att\_val}(a_i, r'_i)$$

$$r' \overset{def}{=} \begin{cases} \sigma'_n.\mathit{ref}(o'_n) & \text{if } \sigma.\mathit{handler}(r) = q \wedge \{a_1, \ldots, a_n\} \neq \{\} \\ \sigma'_0.\mathit{ref}(o'_0) & \text{if } \sigma.\mathit{handler}(r) = q \wedge \{a_1, \ldots, a_n\} = \{\} \\ r & \text{otherwise} \end{cases}$$

$$w' \overset{def}{=} \begin{cases} w'_n & \text{if } \sigma.\mathit{handler}(r) = q \wedge \{a_1, \ldots, a_n\} \neq \{\} \\ w'_0 & \text{if } \sigma.\mathit{handler}(r) = q \wedge \{a_1, \ldots, a_n\} = \{\} \\ w.\mathit{add}(r, r) & \text{otherwise} \end{cases}$$

$$\sigma' \overset{def}{=} \begin{cases} \sigma'_n.\mathit{update\_obj}(\sigma'_n.\mathit{ref}(o'_0), o'_n) & \text{if } \sigma.\mathit{handler}(r) = q \wedge \{a_1, \ldots, a_n\} \neq \{\} \\ \sigma'_0 & \text{if } \sigma.\mathit{handler}(r) = q \wedge \{a_1, \ldots, a_n\} = \{\} \\ \sigma & \text{otherwise} \end{cases}$$

A.25 *STATE* facade: environments

A.25.1 Queries

$envs\colon STATE \to PROC \nrightarrow STACK\langle ENV\rangle$

$\quad \sigma.envs(p)$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\quad$ **axioms**

$\qquad \sigma.envs(p) = \sigma.store.envs(p)$

A.25.2 Commands

$push\_env\_with\_feature\colon STATE \to PROC \nrightarrow FEATURE \to REF$
$\qquad\qquad \to TUPLE\langle REF, \ldots, REF\rangle \nrightarrow STATE$

$\quad \sigma.push\_env\_with\_feature(p, f, r_0, (r_1, \ldots, r_n))$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\qquad f.formals.count = n$

$\qquad \forall i \in \{0, \ldots, n\}\colon r_i \neq void \Rightarrow \sigma.heap.refs.has(r_i)$

$\quad$ **axioms**

$\qquad \sigma.push\_env\_with\_feature(p, f, r_0, (r_1, \ldots, r_n)) =$

$\qquad\quad \sigma.set(\sigma.regions, \sigma.heap, \sigma.store.push\_env(p, e))$

$\qquad$ **where**

$$
\begin{aligned}
w \overset{def}{=}\ & \textbf{new}\ ENV.make \\
& .update(f.formals(1).name, r_1)\ \ldots \\
& .update(f.formals(n).name, r_n) \\
& .update(f.locals(1).name, void)\ \ldots \\
& .update(f.locals(f.locals.count).name, void) \\
& .update(current.name, r_0)
\end{aligned}
$$

$$
e \overset{def}{=} \begin{cases} w & \text{if } f \in PROCEDURE \\ w.update(result.name, void) & \text{if } f \in FUNCTION \end{cases}
$$

$pop\_env\colon STATE \to PROC \nrightarrow STATE$

$\quad \sigma.pop\_env(p)$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\qquad \neg\sigma.store.envs(p).empty$

$\quad$ **axioms**

$\qquad \sigma.pop\_env(p) = \sigma.set(\sigma.regions, \sigma.heap, \sigma.store.pop\_env(p))$

A.26 *STATE* facade: writing and reading values

A.26.1 Queries

$val\colon STATE \to PROC \nrightarrow NAME \nrightarrow REF \cup PROC$

$\quad \sigma.val(p, n)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad \neg\sigma.store.envs(p).empty$

$\quad\quad e.names.has(current.name)$

$\quad\quad e.names.has(n) \vee \exists a \in o.class\_type.attributes\colon a.name = n$

$\quad\quad$ **where**

$\quad\quad\quad e \overset{def}{=} \sigma.store.envs(p).top$

$\quad\quad\quad o \overset{def}{=} \sigma.heap.ref\_obj(e.val(current.name))$

$\quad$ **axioms**

$$\sigma.val(p, n) = \begin{cases} \text{if } \exists a \in o.class\_type.attributes\colon a.name = n \\ \quad o.att\_val(a) \\ \text{otherwise} \\ \quad \sigma.store.envs(p).top.val(n) \end{cases}$$

$\quad\quad$ **where**

$\quad\quad\quad o \overset{def}{=} \sigma.heap.ref\_obj(\sigma.store.envs(p).top.val(current.name))$

$\quad\quad\quad a \overset{def}{=} o.class\_type.feature\_by\_name(n)$

A.26.2 Commands

$set\_val\colon STATE \to PROC \nrightarrow NAME \nrightarrow REF \cup PROC \nrightarrow STATE$

$\quad \sigma.set\_val(p, n, v)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad \neg\sigma.store.envs(p).empty \wedge \sigma.store.envs(p).top.names.has(current.name)$

$\quad\quad v \in REF \wedge v \neq void \Rightarrow \sigma.heap.refs.has(v)$

$\quad\quad v \in PROC \Rightarrow \sigma.regions.procs.has(v)$

$\quad$ **axioms**

$$\sigma.set\_val(p, n, v) = \begin{cases} \text{if } \exists a \in o.class\_type.attributes\colon a.name = n \\ \quad \sigma.update\_obj(\sigma.heap.ref(o), o.set\_att\_val(a, v)) \\ \text{otherwise} \\ \quad \sigma.set(\sigma.regions, \sigma.heap, s) \end{cases}$$

$\quad\quad$ **where**

$\quad\quad\quad o \overset{def}{=} \sigma.heap.ref\_obj(\sigma.store.envs(p).top.val(current.name))$

$\quad\quad\quad a \overset{def}{=} o.class\_type.feature\_by\_name(n)$

$\quad\quad\quad s \overset{def}{=} \sigma.store.pop\_env(p).push\_env(p, \sigma.store.envs(p).top.update(n, v))$

A.27 *STATE* facade: once routines

A.27.1 Queries

$fresh\colon STATE \to PROC \nrightarrow ID \nrightarrow BOOLEAN$
  $\sigma.fresh(p, i)$ **require**
    $\sigma.regions.procs.has(p)$
  axioms
    $\sigma.fresh(p, i) = \sigma.heap.fresh(p, i)$
$stable\colon STATE \to PROC \nrightarrow ID \nrightarrow BOOLEAN$
  $\sigma.stable(p, i)$ **require**
    $\sigma.regions.procs.has(p)$
    $\neg\sigma.heap.fresh(p, i)$
  axioms
    $\sigma.stable(p, i) = \sigma.heap.stable(p, i)$
$stabilizer\colon STATE \to PROC \nrightarrow ID \nrightarrow PROC$
  $\sigma.stabilizer(p, i)$ **require**
    $\sigma.regions.procs.has(p)$
    $\neg\sigma.heap.fresh(p, i)$
  axioms
    $\sigma.stabilizer(p, i) = \sigma.heap.stabilizer(p, i)$
$has\_failed\colon STATE \to PROC \nrightarrow ID \nrightarrow BOOLEAN$
  $\sigma.has\_failed(p, i)$ **require**
    $\sigma.regions.procs.has(p)$
    $\neg\sigma.heap.fresh(p, i)$
  axioms
    $\sigma.has\_failed(p, i) = \sigma.heap.has\_failed(p, i)$
$once\_result\colon STATE \to PROC \nrightarrow ID \nrightarrow REF$
  $\sigma.once\_result(p, i)$ **require**
    $\sigma.regions.procs.has(p)$
    $\neg\sigma.heap.fresh(p, i)$
  axioms
    $\sigma.once\_result(p, i) = \sigma.heap.once\_result(p, i)$

A.27.2 Commands

$set\_once\_func\_not\_fresh\colon STATE \to PROC \nrightarrow FEATURE \nrightarrow BOOLEAN \to BOOLEAN \to REF \nrightarrow STATE$

$\sigma.set\_once\_func\_not\_fresh(p, f, st, hf, r)$ **require**

$\sigma.regions.procs.has(p)$

$f \in FUNCTION \wedge f.is\_once$

$r \neq void \Rightarrow \sigma.heap.refs.has(r)$

axioms

$\sigma.set\_once\_func\_not\_fresh(p, f, st, hf, r) = \sigma.set(\sigma.regions, \sigma.heap.set\_once\_func\_not\_fresh(p, f, st, hf, r), \sigma.store)$

$set\_once\_proc\_not\_fresh\colon STATE \to PROC \nrightarrow FEATURE \nrightarrow BOOLEAN \to BOOLEAN \to STATE$

$\sigma.set\_once\_proc\_not\_fresh(p, f, st, hf)$ **require**

$\sigma.regions.procs.has(p)$

$f \in PROCEDURE \wedge f.is\_once$

axioms

$\sigma.set\_once\_proc\_not\_fresh(p, f, st, hf) = \sigma.set(\sigma.regions, \sigma.heap.set\_once\_proc\_not\_fresh(p, f, st, hf), \sigma.store)$

$set\_once\_rout\_fresh\colon STATE \to PROC \to FEATURE \nrightarrow STATE$

$\sigma.set\_once\_rout\_fresh(p, f)$ **require**

$\sigma.regions.procs.has(p)$

$f.is\_once$

axioms

$\sigma.set\_once\_rout\_fresh(p, f) = \sigma.set(\sigma.regions, \sigma.heap.set\_once\_rout\_fresh(p, f), \sigma.store)$

A.28 *STATE* facade: locks

A.28.1 Queries

$rq\_locked\colon STATE \to PROC \nrightarrow BOOLEAN$

$\sigma.rq\_locked(p)$ **require**

$\sigma.regions.procs.has(p)$

axioms

$\sigma.rq\_locked(p) = \sigma.regions.rq\_locked(p)$

$rq\_locks\colon STATE \to PROC \nrightarrow SET\langle PROC\rangle$

$\quad \sigma.rq\_locks(p)$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\quad$ **axioms**

$\qquad \sigma.rq\_locks(p) =$

$\qquad\quad \sigma.regions.obtained\_rq\_locks(p).flat \cup \sigma.regions.retrieved\_rq\_locks(p).flat$

$cs\_locks\colon STATE \to PROC \nrightarrow SET\langle PROC\rangle$

$\quad \sigma.cs\_locks(p)$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\quad$ **axioms**

$\qquad \sigma.cs\_locks(p) =$

$\qquad\quad \{\sigma.regions.obtained\_cs\_lock(p)\} \cup \sigma.regions.retrieved\_cs\_locks(p).flat$

$passed\colon STATE \to PROC \nrightarrow BOOLEAN$

$\quad \sigma.passed(p)$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\quad$ **axioms**

$\qquad \sigma.passed(p) = \sigma.regions.passed(p)$

A.28.2 Commands

$lock\_rqs\colon STATE \to PROC \nrightarrow SET\langle PROC\rangle \nrightarrow STATE$

$\quad \sigma.lock\_rqs(p, \bar{l})$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\qquad \forall x \in \bar{l}\colon \sigma.regions.procs.has(x)$

$\qquad \forall x \in \bar{l}\colon \sigma.regions.rq\_locked(x) = false$

$\quad$ **axioms**

$\qquad \sigma.lock\_rqs(p, \bar{l}) = \sigma.set(\sigma.regions.lock\_rqs(p, \bar{l}), \sigma.heap, \sigma.store)$

$pop\_obtained\_rq\_locks\colon STATE \to PROC \nrightarrow STATE$

$\quad \sigma.pop\_obtained\_rq\_locks(p)$ **require**

$\qquad \sigma.regions.procs.has(p)$

$\qquad \neg\sigma.regions.obtained\_rq\_locks(p).empty$

$\qquad \sigma.regions.passed(p) = false$

$\quad$ **axioms**

$\qquad \sigma.pop\_obtained\_rq\_locks(p) =$

$\qquad\quad \sigma.set(\sigma.regions.pop\_obtained\_rq\_locks(p), \sigma.heap, \sigma.store)$

$unlock\_rq\colon STATE \rightarrow PROC \nrightarrow STATE$
  $\sigma.unlock\_rq(p)$ **require**
    $\sigma.regions.procs.has(p)$
    $\sigma.regions.rq\_locked(p) = true$
    $\forall q \in \sigma.regions.procs\colon \neg\sigma.regions.obtained\_rq\_locks(q).flat.has(p)$
  axioms
    $\sigma.unlock\_rq(p) = \sigma.set(\sigma.regions.unlock\_rq(p), \sigma.heap, \sigma.store)$

$pass\_locks\colon STATE \rightarrow PROC \nrightarrow PROC \nrightarrow TUPLE\langle SET\langle PROC\rangle, SET\langle PROC\rangle\rangle$
    $\nrightarrow STATE$
  $\sigma.pass\_locks(p, q, (\overline{l_r}, \overline{l_c}))$ **require**
    $\sigma.regions.procs.has(p) \wedge \sigma.regions.procs.has(q)$
    $\forall x \in \overline{l_r}\colon\ \sigma.regions.obtained\_rq\_locks(p).flat.has(x) \vee$
      $\sigma.regions.retrieved\_rq\_locks(p).flat.has(x)$
    $\forall x \in \overline{l_c}\colon\ x = \sigma.regions.obtained\_cs\_lock(p) \vee$
      $\sigma.regions.retrieved\_cs\_locks(p).flat.has(x)$
    $\neg\overline{l_r}.empty \vee \neg\overline{l_c}.empty \Rightarrow \neg\sigma.regions.passed(p)$
  axioms
    $\sigma.pass\_locks(p, q, (\overline{l_r}, \overline{l_c})) =$
      $\sigma.set(\sigma.regions.pass\_locks(p, q, (\overline{l_r}, \overline{l_c})), \sigma.heap, \sigma.store)$

$revoke\_locks\colon STATE \rightarrow PROC \nrightarrow PROC \nrightarrow STATE$
  $\sigma.revoke\_locks(p, q)$ **require**
    $\sigma.regions.procs.has(p) \wedge \sigma.regions.procs.has(q)$
    $\neg\sigma.regions.retrieved\_rq\_locks(q).empty \wedge$
      $\neg\sigma.regions.retrieved\_cs\_locks(q).empty$
    $\sigma.regions.retrieved\_rq\_locks(q).top \subseteq$
      $\sigma.regions.obtained\_rq\_locks(p).flat \cup \sigma.regions.retrieved\_rq\_locks(p).flat$
    $\sigma.regions.retrieved\_cs\_locks(q).top \subseteq$
      $\{\sigma.regions.obtained\_cs\_lock(p)\} \cup \sigma.regions.retrieved\_cs\_locks(p).flat$
    $\sigma.regions.retrieved\_rq\_locks(q).top \cup \sigma.regions.retrieved\_cs\_locks(q).top \neq \{\} \Rightarrow$
      $\sigma.regions.passed(p) = true$
    $\sigma.regions.passed(q) = false$
  axioms
    $\sigma.revoke\_locks(p, q) = \sigma.set(\sigma.regions.revoke\_locks(p, q), \sigma.heap, \sigma.store)$

A.29 *STATE* facade: failures

A.29.1 Queries

```
$has\_failed : STATE \rightarrow PROC \nrightarrow BOOLEAN$
    $\sigma.has\_failed(p)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.has\_failed(p) = \sigma.regions.has\_failed(p)$
```

A.29.2 Commands

```
$set\_failed : STATE \rightarrow PROC \nrightarrow BOOLEAN \rightarrow STATE$
    $\sigma.set\_failed(p, hf)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.set\_failed(p, hf) = \sigma.set(\sigma.regions.set\_failed(p, hf), \sigma.heap, \sigma.store)$
```

A.30 *STATE* facade: passive processors

A.30.1 Queries

```
$passive : STATE \rightarrow PROC \nrightarrow BOOLEAN$
    $\sigma.passive(p)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.passive(p) = \sigma.regions.passive(p)$
$executes\_for : STATE \rightarrow PROC \nrightarrow PROC$
    $\sigma.executes\_for(p)$ **require**
        $\sigma.regions.procs.has(p)$
    **axioms**
        $\sigma.executes\_for(p) = \sigma.regions.executes\_for(p)$
```

A.30.2 Commands

$set\_passive\colon STATE \rightarrow PROC \nrightarrow BOOLEAN \rightarrow STATE$

$\quad \sigma.set\_passive(p, ip)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad ip \Rightarrow \sigma.regions.executes\_for(p) = p$

$\quad\quad \neg ip \Rightarrow \neg\exists q \in \sigma.regions.procs\colon (q \neq p \wedge \sigma.regions.executes\_for(q) = p)$

$\quad$ **axioms**

$\quad\quad \sigma.set\_passive(p, ip) = \sigma.set(\sigma.regions.set\_passive(p, ip), \sigma.heap, \sigma.store)$

$set\_executes\_for\colon STATE \rightarrow PROC \nrightarrow PROC \nrightarrow STATE$

$\quad \sigma.set\_executes\_for(p, q)$ **require**

$\quad\quad \sigma.regions.procs.has(p)$

$\quad\quad \sigma.regions.procs.has(q)$

$\quad\quad p \neq q \Rightarrow \neg\sigma.regions.passive(p) \wedge \sigma.regions.passive(q)$

$\quad$ **axioms**

$\quad\quad \sigma.set\_executes\_for(p, q) = \sigma.set(\sigma.regions.set\_executes\_for(p, q), \sigma.heap, \sigma.store)$

A.31 Initial configuration

A program *prg* has the following initial configuration:

$$\sigma \overset{def}{=} \textbf{new}\ STATE.make$$
$$\sigma' \overset{def}{=} \sigma.add\_proc(\sigma.new\_proc) \quad p \overset{def}{=} \sigma'.last\_added\_proc$$
$$\sigma'' \overset{def}{=} \sigma'.add\_proc(\sigma'.new\_proc) \quad q \overset{def}{=} \sigma''.last\_added\_proc$$
$$\sigma''' \overset{def}{=} \sigma''.add\_obj(q, \sigma''.new\_obj(prg.settings.root\_class)) \quad r \overset{def}{=} \sigma'''.last\_added\_obj$$

$\langle p :: \texttt{lock}(\{q\});$

$\quad \texttt{call}(r, prg.settings.root\_procedure, ());$

$\quad \texttt{issue}(q, \texttt{unlock\_rq});$

$\quad \texttt{pop\_obtained\_locks} \mid q ::, \sigma'''\rangle$

A.32 Issuing mechanism

Issue (non-separate)

$$\frac{q = p \qquad \neg\sigma.passed(p) \wedge \sigma.cs\_locks(p).has(q)}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p, \sigma\rangle \rightarrow \langle p :: s_w; s_p, \sigma\rangle}$$

Issue (separate)

$$\frac{q \neq p \wedge \neg\sigma.passed(q) \quad \neg\sigma.passed(p) \wedge \sigma.rq\_locks(p).has(q)}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \to \langle p :: s_p \mid q :: s_q; s_w, \sigma\rangle}$$

Issue (separate callback)

$$\frac{q \neq p \wedge \sigma.passed(q) \wedge \neg\sigma.passed(p) \wedge \sigma.cs\_locks(p).has(q) \quad \neg\sigma.passed(p) \wedge \sigma.cs\_locks(p).has(q)}{\Gamma \vdash \langle p :: \texttt{issue}(q, s_w); s_p \mid q :: s_q, \sigma\rangle \to \langle p :: s_p \mid q :: s_w; s_q, \sigma\rangle}$$

A.33 Notification mechanism

Processors communicate over channels of type *CHANNEL*. A processor creates a new channel a when using a transition rule with "a is fresh" in the premise.

Wait for result (non-separate)

$$\frac{}{\Gamma \vdash \langle p :: \texttt{result}(a, r); s_w; \texttt{wait}(a, p); s_p, \sigma\rangle \to \langle p :: s_w; s_p[r/a.data], \sigma\rangle}$$

Wait for notify (non-separate)

$$\frac{}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{notify}(a, hf); s_w; \texttt{wait}(a, p); s_p, \sigma\rangle \to \\ \quad \langle p :: s_w; \texttt{provided}\ hf\ \texttt{then fail else nop end}; s_p, \sigma\rangle\end{array}}$$

Wait for result (separate)

$$\frac{}{\Gamma \vdash \langle p :: s_w; \texttt{wait}(a, q); s_p \mid q :: \texttt{result}(a, r); s_q, \sigma\rangle \to \langle p :: s_w; s_p[r/a.data] \mid q :: s_q, \sigma\rangle}$$

Wait for notify (separate)

$$\frac{}{\begin{array}{l}\Gamma \vdash \langle p :: s_w; \texttt{wait}(a, q); s_p \mid q :: \texttt{notify}(a, hf); s_q, \sigma\rangle \to \\ \quad \langle p :: s_w; \texttt{provided}\ hf\ \texttt{then fail else nop end}; s_p \mid q :: s_q, \sigma\rangle\end{array}}$$

Wait for failure (separate)

$$\frac{\sigma.has\_failed(q)}{\Gamma \vdash \langle p :: \texttt{wait}(a, q); s_p \mid q :: s_q, \sigma\rangle \to \langle p :: \texttt{fail}; s_p \mid q :: s_q, \sigma\rangle}$$

A.34 Locking and unlocking mechanism

Lock

$$\frac{\begin{array}{l}\neg\exists q_i \in \{q_1, \ldots, q_m\} \colon \sigma.rq\_locked(q_i) \\ \sigma' \overset{def}{=} \sigma.lock\_rqs(p, \{q_1, \ldots, q_m\})\end{array}}{\Gamma \vdash \langle p :: \texttt{lock}(\{q_1, \ldots, q_m\}); s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Unlock request queue

$$\frac{\begin{array}{l}\sigma.rq\_locked(p) \\ \forall q \in \sigma.procs \colon \neg\sigma.rq\_locks(q).has(p) \\ \sigma' \overset{def}{=} \sigma.unlock\_rq(p).set\_failed(p, false)\end{array}}{\Gamma \vdash \langle p :: \texttt{unlock\_rq}; s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Pop obtained locks

$$\frac{\sigma' \overset{def}{=} \sigma.pop\_obtained\_rq\_locks(p)}{\Gamma \vdash \langle p :: \texttt{pop\_obtained\_locks}; s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

A.35 Writing and reading mechanism

Write

$$\frac{\begin{array}{l}(\sigma', v') \overset{def}{=} \begin{cases}\text{if} \\ \quad ce \wedge \\ \quad v \in REF \wedge v \neq void \wedge \sigma.ref\_obj(v).class\_type.is\_exp \wedge \\ \quad \sigma.handler(v) = \sigma.executes\_for(p) \\ \text{then} \\ \quad\quad (\sigma^*, \sigma^*.last\_added\_obj) \\ \quad\quad\quad \textbf{where} \\ \quad\quad\quad\quad \sigma^* \overset{def}{=} \sigma.add\_obj(\sigma.executes\_for(p), \sigma.ref\_obj(v).copy) \\ \text{otherwise} \\ \quad\quad (\sigma, v)\end{cases} \\ \sigma'' \overset{def}{=} \sigma'.set\_val(p, b.name, v')\end{array}}{\Gamma \vdash \langle p :: \texttt{write}(b, v, ce); s_p, \sigma\rangle \to \langle p :: s_p, \sigma''\rangle}$$

Read

$$\frac{}{\Gamma \vdash \langle p :: \texttt{read}(b, a); s_p, \sigma\rangle \to \langle p :: s_p[\sigma.val(p, b.name)/a.data], \sigma\rangle}$$

A.36 Flow control mechanism

Conditional (true)

$$\dfrac{y \overset{def}{=} \begin{cases} v & \text{if } v \in \mathit{BOOLEAN} \\ \sigma.\mathit{ref\_obj}(v).\mathit{att\_val}(\mathit{item}) & \text{if } v \in \mathit{REF} \wedge \sigma.\mathit{ref\_obj}(v).\mathit{class\_type} = \mathit{boolean} \\ \mathit{false} & \text{otherwise} \end{cases} \qquad y = \mathit{true}}{\Gamma \vdash \langle p :: \texttt{provided}\ v\ \texttt{then}\ s_t\ \texttt{else}\ s_f\ \texttt{end}; s_p, \sigma\rangle \rightarrow \langle p :: s_t; s_p, \sigma\rangle}$$

Conditional (false)

$$\dfrac{y \overset{def}{=} \begin{cases} v & \text{if } v \in \mathit{BOOLEAN} \\ \sigma.\mathit{ref\_obj}(v).\mathit{att\_val}(\mathit{item}) & \text{if } v \in \mathit{REF} \wedge \sigma.\mathit{ref\_obj}(v).\mathit{class\_type} = \mathit{boolean} \\ \mathit{true} & \text{otherwise} \end{cases} \qquad y = \mathit{false}}{\Gamma \vdash \langle p :: \texttt{provided}\ v\ \texttt{then}\ s_t\ \texttt{else}\ s_f\ \texttt{end}; s_p, \sigma\rangle \rightarrow \langle p :: s_f; s_p, \sigma\rangle}$$

Skip

$$\dfrac{}{\Gamma \vdash \langle p :: \texttt{nop}; s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma\rangle}$$

A.37 Failure anchor mechanism

Fail (without failure anchor overwrite)

$$\dfrac{\forall i \in 1, \ldots, n \colon \neg s_i \in \mathit{FAILURE\_ANCHOR}}{\begin{array}{l} \Gamma \vdash \langle p :: \texttt{fail}; \\ \qquad s_1; \ldots; s_n; \\ \qquad \texttt{failure\_anchor}\ \mathit{false} \\ \qquad\quad \texttt{final}\ s_{\mathit{final}} \\ \qquad\quad \texttt{normal}\ s_{\mathit{normal}} \\ \qquad\quad \texttt{rescue}\ \mathit{rescue}\ s_{\mathit{rescue}} \\ \qquad\quad \texttt{retry}\ s_{\mathit{retry}} \\ \qquad\quad \texttt{failure}\ s_{s_{\mathit{failure}}} \\ \qquad \texttt{end}; \\ \qquad s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{fail}\ \texttt{rescue}\ \mathit{rescue}\ s_{\mathit{rescue}}\ \texttt{retry}\ s_{\mathit{retry}}\ \texttt{failure}\ s_{\mathit{failure}}\ \texttt{end}; \\ \qquad s_p, \sigma\rangle \end{array}}$$

Fail (with failure anchor overwrite)

$$\frac{\forall i \in 1, \ldots, n \colon \neg s_i \in \mathit{FAILURE\_ANCHOR}}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{fail}\ \texttt{rescue}\ \mathit{rescue}'\ s'_{rescue}\ \texttt{retry}\ s'_{retry}\ \texttt{failure}\ s'_{failure}\ \texttt{end};\\ \qquad s_1; \ldots; s_n;\\ \qquad \texttt{failure\_anchor}\ \mathit{false}\\ \qquad\quad \texttt{final}\ s_{final}\\ \qquad\quad \texttt{normal}\ s_{normal}\\ \qquad\quad \texttt{rescue}\ \mathit{rescue}\ s_{rescue}\\ \qquad\quad \texttt{retry}\ s_{retry}\\ \qquad\quad \texttt{failure}\ s_{s_{failure}}\\ \qquad \texttt{end};\\ \qquad s_p, \sigma\rangle \rightarrow\\ \quad \langle p :: \texttt{failure\_anchor}\ \mathit{true}\\ \qquad\quad \texttt{final}\ s_{final}\\ \qquad\quad \texttt{normal}\ s_{normal}\\ \qquad\quad \texttt{rescue}\ \mathit{rescue}'\ s'_{rescue}\\ \qquad\quad \texttt{retry}\ s'_{retry}\\ \qquad\quad \texttt{failure}\ s'_{failure}\\ \qquad \texttt{end};\\ \qquad s_p, \sigma\rangle\end{array}}$$

Retry instruction

$$\frac{\forall i \in \{1, \ldots, n\} \colon \neg s_i \in \mathit{FAILURE\_ANCHOR}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{retry};\\ \qquad s_1; \ldots; s_n;\\ \qquad \texttt{failure\_anchor}\ \mathit{true}\\ \qquad\quad \texttt{final}\ s_{final}\\ \qquad\quad \texttt{normal}\ s_{normal}\\ \qquad\quad \texttt{rescue}\ \mathit{false}\ s_{rescue}\\ \qquad\quad \texttt{retry}\ s_{retry}\\ \qquad\quad \texttt{failure}\ s_{s_{failure}}\\ \qquad \texttt{end};\\ \qquad s_p, \sigma\rangle \rightarrow\\ \quad \langle p :: s_{retry};\\ \qquad \texttt{failure\_anchor}\ \mathit{false}\\ \qquad\quad \texttt{final}\ s_{final}\\ \qquad\quad \texttt{normal}\ s_{normal}\\ \qquad\quad \texttt{rescue}\ \mathit{rescue}\ \mathit{true}\\ \qquad\quad \texttt{retry}\ s_{retry}\\ \qquad\quad \texttt{failure}\ s_{failure}\\ \qquad \texttt{end};\\ \qquad s_p, \sigma\rangle\end{array}}$$

Failure anchor

```
────────────────────────────────────────
$\Gamma \vdash \langle p$ :: failure_anchor $failure$
            final $s_{final}$
            normal $s_{normal}$
            rescue $rescue$ $s_{rescue}$
            retry $s_{retry}$
            failure $s_{failure}$
        end;
        $s_p, \sigma\rangle \rightarrow$
    $\langle p$ :: provided $\neg failure$ then
            $s_{final}$; $s_{normal}$
        else
            provided $rescue$ then
                $s_{rescue}$;
                failure_anchor $failure$
                    final $s_{final}$
                    normal $s_{normal}$
                    rescue $false$ $s_{rescue}$
                    retry $s_{retry}$
                    failure $s_{failure}$
                end
            else
                $s_{final}$; $s_{failure}$
            end
        end;
        $s_p, \sigma\rangle$
```

A.38 Entity and literal expressions

Entity expression

$$\frac{e \in ENTITY \qquad a' \text{ is fresh}}{\Gamma \vdash \langle p :: \texttt{eval}(a, e); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{read}(e, a'); \texttt{result}(a, a'.data); s_p, \sigma\rangle}$$

Literal expression

$$\frac{\begin{array}{l} e \in \mathit{LITERAL} \\ \sigma' \overset{def}{=} \begin{cases} \sigma & \text{if } e = \textbf{Void} \\ \sigma.\mathit{add\_obj}(\sigma.\mathit{executes\_for}(p), \mathit{obj}(e)) & \text{otherwise} \end{cases} \\ r \overset{def}{=} \begin{cases} \mathit{void} & \text{if } e = \textbf{Void} \\ \sigma'.\mathit{last\_added\_obj} & \text{otherwise} \end{cases} \end{array}}{\Gamma \vdash \langle p :: \mathtt{eval}(a, e); s_p, \sigma\rangle \to \langle p :: \mathtt{result}(a, r); s_p, \sigma'\rangle}$$

A.39 Feature calls

Command instruction

$$\frac{\forall i \in \{0, \ldots, n\}\colon\ a_i \text{ is fresh}}{\begin{array}{l} \Gamma \vdash \langle p :: e_0.f(e_1, \ldots, e_n); s_p, \sigma\rangle \to \\ \quad \langle p :: \mathtt{eval}(a_0, e_0); \mathtt{eval}(a_1, e_1); \ldots; \mathtt{eval}(a_n, e_n); \\ \qquad \mathtt{wait}(a_0, p); \mathtt{wait}(a_1, p); \ldots; \mathtt{wait}(a_n, p); \\ \qquad \mathtt{call}(a_0.\mathit{data}, f, (a_1.\mathit{data}, \ldots, a_n.\mathit{data})); \\ \qquad s_p, \sigma\rangle \end{array}}$$

Query expression

$$\frac{\begin{array}{l} \forall i \in \{0, \ldots, n\}\colon\ a_i \text{ is fresh} \\ a' \text{ is fresh} \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \mathtt{eval}(a, e_0.f(e_1, \ldots, e_n)); s_p, \sigma\rangle \to \\ \quad \langle p :: \mathtt{eval}(a_0, e_0); \mathtt{eval}(a_1, e_1); \ldots; \mathtt{eval}(a_n, e_n); \\ \qquad \mathtt{wait}(a_0, p); \mathtt{wait}(a_1, p); \ldots; \mathtt{wait}(a_n, p); \\ \qquad \mathtt{call}(a', a_0.\mathit{data}, f, (a_1.\mathit{data}, \ldots, a_n.\mathit{data})); \\ \qquad \mathtt{result}(a, a'.\mathit{data}); \\ \qquad s_p, \sigma\rangle \end{array}}$$

Call runtime (raise)

$$\frac{}{\Gamma \vdash \langle p :: \mathtt{call}(r_0, \mathit{raise}, ()); s_p, \sigma\rangle \to \langle p :: \mathtt{fail}; s_p, \sigma\rangle}$$

Call runtime (set passive)

$$\frac{\begin{array}{l} \neg\sigma.\mathit{rq\_locked}(\sigma.\mathit{handler}(r_1)) \\ \sigma' \overset{def}{=} \sigma.\mathit{set\_passive}(\sigma.\mathit{handler}(r_1), \mathit{true}) \end{array}}{\Gamma \vdash \langle p :: \mathtt{call}(r_0, \mathit{set\_passive}, (r_1)); s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Call runtime (set active)

$$\frac{\begin{array}{l}\neg\sigma.\mathit{rq\_locked}(\sigma.\mathit{handler}(r_1))\\ \sigma' \overset{def}{=} \sigma.\mathit{set\_passive}(\sigma.\mathit{handler}(r_1), \mathit{false})\end{array}}{\Gamma \vdash \langle p :: \texttt{call}(r_0, \mathit{set\_active}, (r_1)); s_p, \sigma\rangle \to \langle p :: s_p, \sigma'\rangle}$$

Call feature (command)

$$\frac{\begin{array}{l}
q \overset{def}{=} \sigma.\mathit{handler}(r_0)\\
g \overset{def}{=} \begin{cases} p & \text{if } \sigma.\mathit{passive}(q)\\ q & \text{otherwise}\end{cases}\\
\bar{l} \overset{def}{=} \begin{cases}
\text{if}\\
\quad g \neq p \wedge \exists i \in \{1,\dots,n\}, z, c \colon \Gamma \vdash f.\mathit{formals}(i) : (!, z, c) \wedge\\
\qquad \sigma.\mathit{ref\_obj}(r_i).\mathit{class\_type}.\mathit{is\_ref} \wedge\\
\qquad (\neg\sigma.\mathit{passed}(p) \wedge (\sigma.\mathit{rq\_locks}(p) \cup \sigma.\mathit{cs\_locks}(p)).\mathit{has}(\sigma.\mathit{handler}(r_i)))\\
\text{then}\\
\qquad (\sigma.\mathit{rq\_locks}(p), \sigma.\mathit{cs\_locks}(p))\\
\text{if}\\
\quad g \neq p \wedge \sigma.\mathit{passed}(g) \wedge \neg\sigma.\mathit{passed}(p) \wedge \sigma.\mathit{cs\_locks}(p).\mathit{has}(g)\\
\text{then}\\
\qquad (\sigma.\mathit{rq\_locks}(p), \sigma.\mathit{cs\_locks}(p))\\
\text{otherwise}\\
\qquad (\{\}, \{\})
\end{cases}\\
\sigma'_0 \overset{def}{=} \sigma\\
\forall i \in \{1,\dots,n\} \colon (\sigma'_i, r'_i) \overset{def}{=}\\
\quad \begin{cases}
\text{if } r_i \neq \mathit{void} \wedge \sigma'_{i-1}.\mathit{ref\_obj}(r_i).\mathit{class\_type}.\mathit{is\_exp} \wedge \sigma'_{i-1}.\mathit{handler}(r_i) \neq q)\\
\quad (\sigma^*, \sigma^*.\mathit{last\_imported\_obj})\\
\qquad \textbf{where}\\
\qquad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.\mathit{import}(q, r_i)\\
\text{if } r_i \neq \mathit{void} \wedge \sigma'_{i-1}.\mathit{ref\_obj}(r_i).\mathit{class\_type}.\mathit{is\_exp} \wedge \sigma'_{i-1}.\mathit{handler}(r_i) = q)\\
\quad (\sigma^*, \sigma^*.\mathit{last\_added\_obj})\\
\qquad \textbf{where}\\
\qquad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.\mathit{add\_obj}(q, \sigma'_{i-1}.\mathit{ref\_obj}(r_i).\mathit{copy})\\
\text{otherwise}\\
\quad (\sigma'_{i-1}, r_i)
\end{cases}\\
a \text{ is fresh}
\end{array}}{\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{call}(r_0, f, (r_1, \dots, r_n)); s_p, \sigma\rangle \to\\
\quad \langle p :: \texttt{issue}(g, \texttt{apply}(a, r_0, f, (r'_1, \dots, r'_n), p, \bar{l}));\\
\qquad \texttt{provided}\ \bar{l} \neq (\{\}, \{\})\ \texttt{then}\ \texttt{wait}(a, g)\ \texttt{else}\ \texttt{nop}\ \texttt{end};\\
\qquad s_p, \sigma'_n\rangle
\end{array}}$$

Call feature (query)

$$
\frac{
\begin{array}{l}
q \overset{def}{=} \sigma.handler(r_0) \\
g \overset{def}{=} \begin{cases} p & \text{if } \sigma.passive(q) \\ q & \text{otherwise} \end{cases} \\
\bar{l} \overset{def}{=} \begin{cases} (\sigma.rq\_locks(p), \sigma.cs\_locks(p)) & \text{if } g \neq p \\ (\{\}, \{\}) & \text{otherwise} \end{cases} \\
\sigma'_0 \overset{def}{=} \sigma \\
\forall i \in \{1, \ldots, n\}\colon (\sigma'_i, r'_i) \overset{def}{=} \\
\quad \begin{cases}
\text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) \neq q \\
\quad (\sigma^*, \sigma^*.last\_imported\_obj) \\
\quad\quad \textbf{where} \\
\quad\quad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.import(q, r_i) \\
\text{if } r_i \neq void \wedge \sigma'_{i-1}.ref\_obj(r_i).class\_type.is\_exp \wedge \sigma'_{i-1}.handler(r_i) = q \\
\quad (\sigma^*, \sigma^*.last\_added\_obj) \\
\quad\quad \textbf{where} \\
\quad\quad\quad \sigma^* \overset{def}{=} \sigma'_{i-1}.add\_obj(q, \sigma'_{i-1}.ref\_obj(r_i).copy) \\
\text{otherwise} \\
\quad (\sigma'_{i-1}, r_i)
\end{cases}
\end{array}
}{
\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{call}(a, r_0, f, (r_1, \ldots, r_n)); s_p, \sigma \rangle \rightarrow \\
\quad \langle p :: \texttt{issue}(g, \texttt{apply}(a, r_0, f, (r'_1, \ldots, r'_n), p, \bar{l})); \texttt{wait}(a, g); s_p, \sigma'_n \rangle
\end{array}
}
$$

A.40 Feature applications

Apply feature (attribute)

$$
\frac{
\begin{array}{l}
w \overset{def}{=} \sigma.executes\_for(p) \\
\sigma' \overset{def}{=} \sigma.set\_executes\_for(p, \sigma.handler(r_0)) \\
f \in ATTRIBUTE \\
\neg\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \sigma'.handler(r_0) = p \\
\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \neg\sigma'.passed(p) \wedge \sigma'.rq\_locks(p).has(\sigma'.handler(r_0)) \\
\sigma'' \overset{def}{=} \sigma'.pass\_locks(q, p, (\bar{l}_r, \bar{l}_c)).push\_env\_with\_feature(p, f, r_0, ()) \\
a' \text{ is fresh}
\end{array}
}{
\begin{array}{l}
\Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma \rangle \rightarrow \\
\quad \langle p :: \texttt{eval}(a', f); \\
\quad\quad \texttt{wait}(a', p); \\
\quad\quad \texttt{return}(a, f, a'.data, q, (\bar{l}_r, \bar{l}_c), w, false, false, false); \\
\quad\quad s_p, \sigma'' \rangle
\end{array}
}
$$

Apply feature (non-once routine or fresh once routine)

$$
\begin{array}{l}
w \overset{def}{=} \sigma.executes\_for(p) \\
\sigma' \overset{def}{=} \sigma.set\_executes\_for(p, \sigma.handler(r_0)) \\
f \in ROUTINE \wedge (f.is\_once \Rightarrow \sigma.fresh(\sigma'.executes\_for(p), f.id)) \\
\neg\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \sigma'.handler(r_0) = p \\
\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \neg\sigma'.passed(p) \wedge \sigma'.rq\_locks(p).has(\sigma'.handler(r_0)) \\
\sigma'' \overset{def}{=} \begin{cases} \text{if } f \in FUNCTION \wedge f.is\_once \\ \quad \sigma'.set\_once\_func\_not\_fresh(\sigma'.executes\_for(p), f, false, false, void) \\ \text{if } f \in PROCEDURE \wedge f.is\_once \\ \quad \sigma'.set\_once\_proc\_not\_fresh(\sigma'.executes\_for(p), f, false, false) \\ \text{otherwise} \quad \sigma' \end{cases} \\
\sigma''' \overset{def}{=} \sigma''.pass\_locks(q, p, (\overline{l_r}, \overline{l_c})).push\_env\_with\_feature(p, f, r_0, (r_1, \dots, r_n)) \\
\overline{g}_{required\_rq\_and\_cs\_locks} \overset{def}{=} \{p\} \cup \\
\quad \{x \in PROC \mid \exists i \in \{1, \dots, n\}, g, c \colon \Gamma \vdash f.formals(i) : (!, g, c) \wedge \\
\quad \sigma'''.ref\_obj(r_i).class\_type.is\_ref \wedge x = \sigma'''.handler(r_i)\} \\
\overline{g}_{required\_cs\_locks} \overset{def}{=} \\
\quad \{x \in \overline{g}_{required\_rq\_and\_cs\_locks} \mid x = p \vee \\
\quad (x \neq p \wedge \sigma'''.passed(x) \wedge \neg\sigma'''.passed(p) \wedge \sigma'''.cs\_locks(p).has(x)\} \\
\overline{g}_{required\_rq\_locks} \overset{def}{=} \overline{g}_{required\_rq\_and\_cs\_locks} \setminus \overline{g}_{required\_cs\_locks} \\
\{g_1, \dots, g_m\} \overset{def}{=} \overline{g}_{missing\_rq\_locks} \overset{def}{=} \{x \in \overline{g}_{required\_rq\_locks} \mid \neg\sigma'''.rq\_locks(p).has(x)\} \\
a' \text{ is fresh} \\
\hline
\Gamma \vdash \langle p :: \mathtt{apply}(a, r_0, f, (r_1, \dots, r_n), q, (\overline{l_r}, \overline{l_c})); s_p, \sigma\rangle \rightarrow \\
\quad \langle p :: \mathtt{check\_pre\_and\_lock}(a, f, q, (\overline{l_r}, \overline{l_c}), \overline{g}_{missing\_rq\_locks}, w); \\
\qquad \mathtt{execute\_body}(f, \overline{g}_{missing\_rq\_locks}); \mathtt{check\_post\_and\_inv}(f); \\
\qquad \mathtt{failure\_anchor}\ false \\
\qquad\quad \mathtt{final} \\
\qquad\qquad \mathtt{issue}(g_1, \mathtt{unlock\_rq}); \dots; \mathtt{issue}(g_m, \mathtt{unlock\_rq}); \\
\qquad\qquad \mathtt{pop\_obtained\_locks} \\
\qquad\quad \mathtt{normal} \\
\qquad\qquad \mathtt{provided}\ f \in FUNCTION\ \mathtt{then} \\
\qquad\qquad\quad \mathtt{read}(result, a'); \mathtt{return}(a, f, a'.data, q, (\overline{l_r}, \overline{l_c}), w, false, true, false) \\
\qquad\qquad \mathtt{else}\ \mathtt{return}(a, f, q, (\overline{l_r}, \overline{l_c}), w, false, true, false)\ \mathtt{end} \\
\qquad\quad \mathtt{rescue}\ true\ \mathtt{execute\_rescue\_clause}(f) \\
\qquad\quad \mathtt{retry}\ \mathtt{execute\_body}(f, \overline{g}_{missing\_rq\_locks}); \mathtt{check\_post\_and\_inv}(f) \\
\qquad\quad \mathtt{failure}\ \mathtt{return}(a, f, q, (\overline{l_r}, \overline{l_c}), w, true, true, false) \\
\qquad \mathtt{end}; s_p, \sigma'''\rangle
\end{array}
$$

Apply feature (not fresh once routine)

$$\frac{\begin{array}{l} w \overset{def}{=} \sigma.executes\_for(p) \\ \sigma' \overset{def}{=} \sigma.set\_executes\_for(p, \sigma.handler(r_0)) \\ f \in ROUTINE \wedge f.is\_once \wedge \neg\sigma'.fresh(\sigma'.executes\_for(p), f.id) \wedge \\ \quad (\sigma'.stable(\sigma'.executes\_for(p), f.id) \vee \\ \quad \sigma'.stabilizer(\sigma'.executes\_for(p), f.id) = \sigma'.executes\_for(p)) \\ \neg\sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \sigma'.handler(r_0) = p \\ \sigma'.passive(\sigma'.handler(r_0)) \Rightarrow \neg\sigma'.passed(p) \wedge \sigma'.rq\_locks(p).has(\sigma'.handler(r_0)) \\ \sigma'' \overset{def}{=} \sigma'.pass\_locks(q, p, (\bar{l}_r, \bar{l}_c)).push\_env\_with\_feature(p, f, r_0, (r_1, \ldots, r_n)) \end{array}}{\Gamma \vdash \langle p :: \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \rightarrow}$$

```
⟨p :: provided $\sigma''.has\_failed(\sigma''.executes\_for(p), f)$ then
        fail
    else
        nop
    end;
    failure_anchor $false$
        final nop
        normal
            provided $f \in FUNCTION$ then
                return($a, f, \sigma''.once\_result(\sigma''.executes\_for(p), f.id),$
                    $q, (\bar{l}_r, \bar{l}_c), w, false, false, false$)
            else
                return($a, f, q, (\bar{l}_r, \bar{l}_c), w, false, false, false$)
            end
        rescue $false$ nop
        retry nop
        failure return($a, f, q, (\bar{l}_r, \bar{l}_c), w, true, false, false$)
    end;
    $s_p, \sigma''$⟩
```

Check precondition and lock

$$targets(e) \overset{def}{=} \begin{cases} \{e_0\} \bigcup_{i=1,\ldots,n} targets(e_i) & \text{if } e = e_0.h(e_1,\ldots,e_n) \\ \{\} & \text{otherwise} \end{cases}$$

$\overline{g} \overset{def}{=} \{x \in \sigma.procs \mid (\exists y, z \in EXPRESSION: \; y \in targets(f.pre)$
$\wedge z = controlling\_entity(\Gamma, y)) \wedge x = \sigma.handler(\sigma.val(p, z.name)) \wedge x \neq p\}$

a is fresh

$cc \overset{def}{=} (p = q \vee f \in FUNCTION \vee (\bar{l}_r, \bar{l}_c) \neq (\{\}, \{\})) \wedge$
$(\overline{g} \subseteq \sigma.rq\_locks(q) \cup \sigma.cs\_locks(q))$

```
Γ ⊢⟨$p$ :: check_pre_and_lock($a, f, q, (\bar{l}_r, \bar{l}_c), \{g_1, \ldots, g_m\}, w$); $s_p, \sigma$⟩ →
   ⟨$p$ :: lock($\{g_1, \ldots, g_m\}$);
         provided $f.has\_pre$ then
            eval($a, f.pre$);
            wait($a, p$);
            provided $a.data$ then
               nop
            else
               provided $cc$ then
                  fail
                     rescue $false$ nop
                     retry nop
                     failure return($a, f, q, (\bar{l}_r, \bar{l}_c), w, true, false, true$)
                  end
               else
                  issue($g_1$, unlock_rq);
                  ...
                  issue($g_m$, unlock_rq);
                  pop_obtained_locks;
                  check_pre_and_lock($a, f, q, (\bar{l}_r, \bar{l}_c), \{g_1, \ldots, g_m\}, w$)
               end
            end
         else
            nop
         end;
         $s_p, \sigma$⟩
```

Execute body

$$\frac{\forall i \in \{1,\ldots,n\}\colon\ a_i \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \mathtt{execute\_body}(f, \{g_1,\ldots,g_n\}); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \mathtt{provided}\ f \in \mathit{FUNCTION} \wedge f.\mathit{is\_once}\ \mathtt{then} \\ \qquad\qquad f.\mathit{body} \\ \qquad\qquad\quad [\mathit{result} := y; \mathtt{set\_not\_fresh}(f)/\mathit{result} := y] \\ \qquad\qquad\quad [\mathbf{create}\ \mathit{result}.y; \mathtt{set\_not\_fresh}(f)/\mathbf{create}\ \mathit{result}.y] \\ \qquad\quad \mathtt{else} \\ \qquad\qquad f.\mathit{body} \\ \qquad\quad \mathtt{end}; \\ \qquad\quad \mathtt{provided}\ f.\mathit{class\_type.program.settings.safe\_mode}\ \mathtt{then} \\ \qquad\qquad \mathtt{issue}(g_1, \mathtt{notify}(a_1, \mathit{false})); \mathtt{wait}(a_1, g_1); \\ \qquad\qquad \ldots; \\ \qquad\qquad \mathtt{issue}(g_n, \mathtt{notify}(a_n, \mathit{false})); \mathtt{wait}(a_n, g_n) \\ \qquad\quad \mathtt{else} \\ \qquad\qquad \mathtt{nop} \\ \qquad\quad \mathtt{end}; \\ \qquad\quad s_p, \sigma\rangle\end{array}}$$

Execute rescue clause

$$\frac{}{\begin{array}{l}\Gamma \vdash \langle p :: \mathtt{execute\_rescue\_clause}(f); s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \mathtt{provided}\ f \in \mathit{FUNCTION} \wedge f.\mathit{is\_once}\ \mathtt{then} \\ \qquad\qquad f.\mathit{rescue\_clause} \\ \qquad\qquad\quad [\mathit{result} := y; \mathtt{set\_not\_fresh}(f)/\mathit{result} := y] \\ \qquad\qquad\quad [\mathbf{create}\ \mathit{result}.y; \mathtt{set\_not\_fresh}(f)/\mathbf{create}\ \mathit{result}.y] \\ \qquad\quad \mathtt{else} \\ \qquad\qquad f.\mathit{rescue\_clause} \\ \qquad\quad \mathtt{end}; \\ \qquad\quad s_p, \sigma\rangle\end{array}}$$

Set not fresh

$$\frac{\begin{array}{l} f \in \mathit{FUNCTION} \wedge f.\mathit{is\_once} \\ \sigma.\mathit{envs}(p).\mathit{top.names.has}(\mathit{result.name}) \\ \sigma' \overset{def}{=} \sigma.\mathit{set\_once\_func\_not\_fresh}(\\ \quad \sigma.\mathit{executes\_for}(p), f, \mathit{false}, \mathit{false}, \sigma.\mathit{val}(p, \mathit{result.name}))\end{array}}{\Gamma \vdash \langle p :: \mathtt{set\_not\_fresh}(f); s_p, \sigma\rangle \rightarrow \langle p :: s_p, \sigma\rangle}$$

Check postcondition and invariant

$$\frac{a_{inv} \text{ is fresh} \quad a_{post} \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{check\_post\_and\_inv}(f);\, s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ f.has\_post\ \texttt{then} \\ \qquad\quad \texttt{eval}(a_{post}, f.post); \texttt{wait}(a_{post}, p); \\ \qquad\quad \texttt{provided}\ a_{post}.data\ \texttt{then nop else fail end} \\ \qquad \texttt{else} \\ \qquad\quad \texttt{nop} \\ \qquad \texttt{end}; \\ \qquad \texttt{provided}\ f.class\_type.has\_inv \wedge f.is\_exported\ \texttt{then} \\ \qquad\quad \texttt{eval}(a_{inv}, f.class\_type.inv); \texttt{wait}(a_{inv}, p); \\ \qquad\quad \texttt{provided}\ a_{inv}.data\ \texttt{then nop else fail end} \\ \qquad \texttt{else} \\ \qquad\quad \texttt{nop} \\ \qquad \texttt{end};\, s_p, \sigma\rangle\end{array}}$$

Return (successful command)

$$\frac{\begin{array}{l}\sigma' \overset{def}{=} \begin{cases} \text{if } f.is\_once \wedge snf \\ \quad \sigma.set\_once\_proc\_not\_fresh(\sigma.executes\_for(p), f, true, false) \\ \text{if } f.is\_once \wedge sf \\ \quad \sigma.set\_once\_rout\_fresh(\sigma.executes\_for(p), f) \\ \text{otherwise} \\ \quad \sigma \end{cases} \\ \sigma'' \overset{def}{=} \sigma'.pop\_env(p).revoke\_locks(q, p).set\_executes\_for(p, w)\end{array}}{\begin{array}{l}\Gamma \vdash \langle p :: \texttt{return}(a, f, q, (\bar{l}_r, \bar{l}_c), w, false, snf, sf);\, s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{provided}\ (\bar{l}_r, \bar{l}_c) \neq (\{\}, \{\})\ \texttt{then notify}(a, false)\ \texttt{else nop end}; \\ \qquad s_p, \sigma''\rangle\end{array}}$$

Return (successful query)

$$g \overset{def}{=} \begin{cases} w & \text{if } p = q \\ \sigma.executes\_for(q) & \text{otherwise} \end{cases}$$

$$(\sigma', r_r') \overset{def}{=} \begin{cases} \text{if } r_r \neq void \wedge \sigma.ref\_obj(r_r).class\_type.is\_exp \wedge \sigma.handler(r_r) \neq g \\ \quad (\sigma^*, \sigma^*.last\_imported\_obj) \\ \quad\quad \textbf{where} \\ \quad\quad\quad \sigma^* \overset{def}{=} \sigma.import(g, r_r) \\ \text{otherwise} \\ \quad (\sigma, r_r) \end{cases}$$

$$\sigma'' \overset{def}{=} \begin{cases} \text{if } f.is\_once \wedge snf \\ \quad \sigma'.set\_once\_func\_not\_fresh(\sigma.executes\_for(p), f, true, false, r_r) \\ \text{if } f.is\_once \wedge sf \\ \quad \sigma'.set\_once\_rout\_fresh(\sigma.executes\_for(p), f) \\ \text{otherwise} \\ \quad \sigma' \end{cases}$$

$$\sigma''' \overset{def}{=} \sigma''.pop\_env(p).revoke\_locks(q, p).set\_executes\_for(p, w)$$

$$\Gamma \vdash \langle p :: \texttt{return}(a, f, r_r, q, (\bar{l}_r, \bar{l}_c), w, false, snf, sf); s_p, \sigma\rangle \rightarrow \langle p :: \texttt{result}(a, r_r'); s_p, \sigma'''\rangle$$

Return (failure)

$$\sigma' \overset{def}{=} \begin{cases} \text{if } f \in FUNCTION \wedge f.is\_once \wedge snf \\ \quad \sigma.set\_once\_func\_not\_fresh(\sigma.executes\_for(p), f, true, true, void) \\ \text{if } f \in PROCEDURE \wedge f.is\_once \wedge snf \\ \quad \sigma.set\_once\_proc\_not\_fresh(\sigma.executes\_for(p), f, true, true) \\ \text{if } f.is\_once \wedge sf \\ \quad \sigma.set\_once\_rout\_fresh(\sigma.executes\_for(p), f) \\ \text{otherwise} \\ \quad \sigma \end{cases}$$

$$\sigma'' \overset{def}{=} \begin{cases} \sigma'.set\_failed(p, true) & \text{if } p \neq q \wedge f \notin FUNCTION \wedge (\bar{l}_r, \bar{l}_c) = (\{\}, \{\}) \\ \sigma' & \text{otherwise} \end{cases}$$

$$\sigma''' \overset{def}{=} \sigma''.pop\_env(p).revoke\_locks(q, p).set\_executes\_for(p, w)$$

```
Γ ⊢⟨$p$ :: return($a, f, q, (\bar{l}_r, \bar{l}_c), w, true, snf, sf$); $s_p, \sigma$⟩ →
   ⟨$p$ :: provided $p = q$ then fail else
            provided $f \notin FUNCTION \wedge (\bar{l}_r, \bar{l}_c) = (\{\}, \{\})$ then
               purge
            else
               notify($a, true$)
            end
         end;
        $s_p, \sigma'''$⟩
```

Purge request queue unlock request

$$\frac{}{\Gamma \vdash \langle p :: \texttt{purge}; \texttt{unlock\_rq}; s_p, \sigma\rangle \to \langle p :: \texttt{unlock\_rq}; s_p, \sigma\rangle}$$

Purge feature request

$$\frac{}{\Gamma \vdash \langle p :: \texttt{purge}; \texttt{apply}(a, r_0, f, (r_1, \ldots, r_n), q, (\bar{l}_r, \bar{l}_c)); s_p, \sigma\rangle \to \langle p :: \texttt{purge}; s_p, \sigma\rangle}$$

Purge notification request

$$\frac{}{\Gamma \vdash \langle p :: \texttt{purge}; \texttt{notify}(a, \mathit{false}); s_p, \sigma\rangle \to \langle p :: \texttt{purge}; s_p, \sigma\rangle}$$

A.41 Creation instructions

Creation instruction (separate)

$$(d, g, c) \overset{def}{=} \mathit{type\_of}(\Gamma, b)$$
$$g = \top$$
$$q \overset{def}{=} \sigma.\mathit{new\_proc}$$
$$o \overset{def}{=} \sigma.\mathit{new\_obj}(c)$$
$$\sigma' \overset{def}{=} \sigma.\mathit{add\_proc}(q).\mathit{add\_obj}(q, o)$$
$$r \overset{def}{=} \sigma'.\mathit{ref}(o)$$
$$a \text{ is fresh}$$

```
Γ ⊢⟨p :: create $b.f(e_1, \ldots, e_n); s_p, \sigma\rangle \to$
   ⟨p :: lock({q});
        write(b, r, false);
        $b.f(e_1, \ldots, e_n)$;
        provided $f.\mathit{class\_type.program.settings.safe\_mode}$ then
           issue(q, notify(a, false)); wait(a, q)
        else
           nop
        end;
        failure_anchor false
           final issue(q, unlock_rq); pop_obtained_locks
           normal nop
           rescue false nop
           retry nop
           failure fail
        end;
        $s_p \mid q$ :: nop, $\sigma'\rangle$
```

Creation instruction (existing explicitly specified processor)

$$(d, g, c) \overset{def}{=} type\_of(\Gamma, b)$$
$$(\exists x \in ENTITY: g =< x >) \vee (\exists x \in ENTITY: g =< x.handler >)$$
$$q \overset{def}{=} \begin{cases} \sigma.val(p, x.name) & \text{if } \exists x \in ENTITY: g =< x > \\ \sigma.handler(\sigma.val(p, x.name)) & \text{if } \exists x \in ENTITY: g =< x.handler > \end{cases}$$
$$\sigma.procs.has(q)$$
$$o \overset{def}{=} \sigma.new\_obj(c)$$
$$\sigma' \overset{def}{=} \sigma.add\_obj(q, o)$$
$$r \overset{def}{=} \sigma'.ref(o)$$
$$w \overset{def}{=} q \neq p \wedge \neg(q \neq p \wedge \sigma'.passed(q) \wedge \neg\sigma'.passed(p) \wedge \sigma'.cs\_locks(p).has(q)) \wedge \neg\sigma'.rq\_locks(p).has(q)$$
a is fresh

```
Γ ⊢⟨p :: create b.f(e_1, ..., e_n); s_p, σ⟩ →
   ⟨p :: provided w then
            lock({q})
         else
            nop
         end;
         write(b, r, false);
         b.f(e_1, ..., e_n);
         provided w then
            provided f.class_type.program.settings.safe_mode then
               issue(q, notify(a, false)); wait(a, q)
            else
               nop
            end;
            failure_anchor false
               final issue(q, unlock_rq); pop_obtained_locks
               normal nop
               rescue false nop
               retry nop
               failure fail
            end
         else
            nop
         end;
         s_p, σ'⟩
```

Creation instruction (non-existing explicitly specified processor)

$$\frac{\begin{array}{l} (d,g,c) \overset{def}{=} \mathit{type\_of}(\Gamma, b) \\ \exists x \in \mathit{ENTITY}:\ g =< x > \\ < x > \overset{def}{=} g \\ \neg\sigma.\mathit{procs}.\mathit{has}(\sigma.\mathit{val}(p, x.\mathit{name})) \\ q \overset{def}{=} \sigma.\mathit{new\_proc} \\ o \overset{def}{=} \sigma.\mathit{new\_obj}(c) \\ \sigma' \overset{def}{=} \sigma.\mathit{add\_proc}(q).\mathit{add\_obj}(q, o) \\ r \overset{def}{=} \sigma'.\mathit{ref}(o) \\ a \text{ is fresh} \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1, \ldots, e_n); s_p, \sigma\rangle \to \\ \quad \langle p :: \mathtt{write}(x, q, \mathit{false}); \\ \qquad \mathtt{lock}(\{q\}); \\ \qquad \mathtt{write}(b, r, \mathit{false}); \\ \qquad b.f(e_1, \ldots, e_n); \\ \qquad \mathtt{provided}\ f.\mathit{class\_type}.\mathit{program}.\mathit{settings}.\mathit{safe\_mode}\ \mathtt{then} \\ \qquad\quad \mathtt{issue}(q, \mathtt{notify}(a, \mathit{false})); \mathtt{wait}(a, q) \\ \qquad \mathtt{else} \\ \qquad\quad \mathtt{nop} \\ \qquad \mathtt{end}; \\ \qquad \mathtt{failure\_anchor}\ \mathit{false} \\ \qquad\quad \mathtt{final}\ \mathtt{issue}(q, \mathtt{unlock\_rq}); \mathtt{pop\_obtained\_locks} \\ \qquad\quad \mathtt{normal}\ \mathtt{nop} \\ \qquad\quad \mathtt{rescue}\ \mathit{false}\ \mathtt{nop} \\ \qquad\quad \mathtt{retry}\ \mathtt{nop} \\ \qquad\quad \mathtt{failure}\ \mathtt{fail} \\ \qquad \mathtt{end}; \\ \qquad s_p \mid q :: \mathtt{nop}, \sigma'\rangle \end{array}}$$

Creation instruction (non-separate)

$$\frac{\begin{array}{l} (d,g,c) \overset{def}{=} \mathit{type\_of}(\Gamma, b) \\ g = \bullet \\ o \overset{def}{=} \sigma.\mathit{new\_obj}(c) \\ \sigma' \overset{def}{=} \sigma.\mathit{add\_obj}(\sigma.\mathit{executes\_for}(p), o) \\ r \overset{def}{=} \sigma'.\mathit{ref}(o) \end{array}}{\begin{array}{l} \Gamma \vdash \langle p :: \textbf{create}\ b.f(e_1, \ldots, e_n); s_p, \sigma\rangle \to \\ \quad \langle p :: \mathtt{write}(b, r, \mathit{false}); \\ \qquad b.f(e_1, \ldots, e_n); \\ \qquad s_p, \sigma'\rangle \end{array}}$$

A.42 Flow control instructions

If instruction (with else)

$$\frac{a \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{if}\ e\ \textbf{then}\ s_t\ \textbf{else}\ s_f\ \textbf{end}; s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a, e); \\ \qquad \texttt{wait}(a, p); \\ \qquad \texttt{provided}\ a.data\ \texttt{then}\ s_t\ \texttt{else}\ s_f\ \texttt{end}; \\ \qquad s_p, \sigma\rangle\end{array}}$$

If instruction (without else)

$$\frac{a \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{if}\ e\ \textbf{then}\ s_t\ \textbf{end}; s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a, e); \\ \qquad \texttt{wait}(a, p); \\ \qquad \texttt{provided}\ a.data\ \texttt{then}\ s_t\ \texttt{else}\ \texttt{nop}\ \texttt{end}; \\ \qquad s_p, \sigma\rangle\end{array}}$$

Loop instruction

$$\frac{a \text{ is fresh}}{\begin{array}{l}\Gamma \vdash \langle p :: \textbf{until}\ e\ \textbf{loop}\ s_l\ \textbf{end}; s_p, \sigma\rangle \rightarrow \\ \quad \langle p :: \texttt{eval}(a, e); \\ \qquad \texttt{wait}(a, p); \\ \qquad \texttt{provided}\ a.data\ \texttt{then}\ \texttt{nop}\ \texttt{else}\ s_l; \textbf{until}\ e\ \textbf{loop}\ s_l\ \textbf{end}\ \texttt{end}; \\ \qquad s_p, \sigma\rangle\end{array}}$$

A.43 Assignment instructions

Assignment instruction

$$\frac{a \text{ is fresh}}{\Gamma \vdash \langle p :: b := e; s_p, \sigma\rangle \rightarrow \langle p :: \texttt{eval}(a, e); \texttt{wait}(a, p); \texttt{write}(b, a.data, true); s_p, \sigma\rangle}$$

A.44 Parallelism

Parallelism

$$\frac{\Gamma \vdash \langle P, \sigma\rangle \rightarrow \langle P', \sigma'\rangle}{\Gamma \vdash \langle P \mid Q, \sigma\rangle \rightarrow \langle P' \mid Q, \sigma'\rangle}$$