Diss. ETH No. 24002

Unified Interference-free Parallel, Concurrent and Distributed Programming

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
Mischael Schill
Master of Science ETH in Computer Science

born on
July 26th, 1984

citizen of
Winterthur, Switzerland and Austria

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Friedemann Mattern, co-examiner
Prof. Dr. Richard Paige, co-examiner
Dr. Piotr Nienaltowski, co-examiner

2016

Acknowledgments.

I want to thank all that helped and supported me during my studies that lead to this work. First I thank my family: my parents, Gottfried and Maria, for supporting me during my studies, and my wife, Simona, for encouraging me to pursue a PhD.

Bertrand Meyer, my supervisor, who welcomed me to his research group, supported me and created a fruitful environment for exciting research. His suggestions, guidance and critique encouraged me made me come up and re-think ideas until perfection.

The people working at the legendary Chair of Software Engineering: Sebastian Nanz, who taught me how to be a researcher. His gentle guidance helped me find my area and bring ideas to paper. Chris Poskitt, who I worked with during my last year, for the many late hours of work and encouragement. Claudia, for doing everything to keep the Chair working, even things that were not her responsibility, and for the many conversations during my last year. Benjamin Morandi, who supervised my master thesis and whetted my interest for doing research. Marco T., with whom I had many interesting and hilarious conversations. Carlo, Marco P., Scott, Stephan, Andrey, Alexey, Max, Martin, Georgiana, Jason, Jiwon, and Juri.

Peter Müller for providing me with a nice place to finish my thesis during the last year, and all the people in CAB H66 for welcoming me.

Lastly, the examiners who took the time to read and evaluate my work: Friedemann Mattern, who kindly agreed to be my second supervisor, Richard Paige and Piotr Nienaltowski.

Funding. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389.

Contents

Abstract

Concurrency, the art of doing many things at the same time, is growing in importance every year. For almost a decade, computers are no longer getting faster the same way they have been before, but they gain the ability to do more work in parallel. Most modern programs and their functionality rely on global distributed infrastructure: communication between computers has become essential, distributed systems are becoming the norm, not the exception.

Although the integration of systems and programs becomes tighter and tighter, two areas are still most often treated as separate: local concurrency and parallelization, versus distribution. While there are concurrency programming models that can operate in both areas, they are often a compromise: they rarely take full advantage of shared memory available in a machine and, at the same time, give the developers inadequate tools to handle complex atomicity violations. The result is that software is often written using an unsafe multi-threading approach coupled with simple message passing for communication between computers.

This work improves the current state of the art in two ways. First by addressing the problem that these models often disregard the dependencies between requests, resulting in atomicity violations similar to data races. Second by reconciling models for concurrency and distribution with shared memory.

For the first, this work extends the SCOOP concurrency model for distributed programming. SCOOP provides interference-freedom to minimize atomicity violations, but is restricted to local computing only. With distribution, failure handling is important: we present a novel approach — compensations s— which are inspired by transactions, to let supplier and clients both contribute to failure mitigation. The core semantics of distributed SCOOP, the result of this endeavor, is described formally using a transition semantics that can be used as a specification for implementations. A prototype of D-SCOOP shows that interference-freedom can be achieved without sacrificing performance.

For the second, this work presents two techniques:

Slicing A technique for handling arrays
Immutable Classes A technique for handling complex immutable data

With our work on integration of shared memory computing and distribution we combine local concurrency and parallelism with distribution to form a general model.

Zusammenfassung

Nebenläufigkeit, die Kunst viele Dinge gleichzeitig zu tun, wird wichtiger von Jahr zu Jahr. Seit fast einem Jahrzeht werden Computer nicht mehr in dem Masse schneller wie davor, aber sie erhalten die Fähigkeit mehr Arbeit parallel erledigen zu können. Die meisten Programme und ihre Funktionalität stützen sich auf eine global verteilte Infrastruktur: Die Kommunikation zwischen Computern wurde essentiell, verteilte Systeme werden zur Norm.

Obwohl die Integration von Systemen und Programm immer enger und enger wird, werden zwei Gebiete häufig immer noch getrennt behandelt: Lokale Nebenläufigkeit und Parallelität gegenüber Verteilten System. Während zwar Programmiermodelle für Nebenläufigkeit existieren, die für beide Gebiete geeignet sind, sind diese häufig ein Kompromiss: Sie ziehen selten das Maximum aus der Verfügbarkeit von gemeinsamen Speicher einer Maschine, und geben gleichzeitig dem Entwickler nur inadäquate Wekzeuge zur Vermeidung von komplexen Atomizitätsverletzungen zur Hand. Das Resultat ist, dass Software oft noch immer mit einem unsicheren Prozessmodell in Kombination mit einfachem Nachrichtenaustausch erstellt werden.

Diese Arbeit bringt den aktuellen Stand der Technik auf zwei Arten vorwärts: Sie nimmt sich des Problems, dass solcherlei Modelle die Abhängigkeiten zwischen Anfragen, welche zu Atomizitätsverletzungen führen, oft ignorieren sowie des Problems der Vereinheitlichung von Nebenläufigkeit und verteilten Systemen mit gemeinsamen Speicher, an.

Für ersteres präsentiert diese Arbeit eine Erweiterung des Scoop Modells für die Verwendung in verteilten Systemen. Scoop verfügt über ein Konzept der Interferenzfreiheit um Atomizitätsverletzungen zu minimieren, aber es ist bis jetzt auf die Verwendung als lokales System beschränkt. In verteilten Systemen ist die Handhabung von Fehlern prioritär, darum präsentieren wir einen neuartigen Anzatz, Kompensationen, inspiriert von Transaktionssystemen, um den Befehlsempfänger sie auch den Befehlssender an der Fehlerbehebung teilhaben zu lassen. Die Kernsemantik von verteiltem Scoop, genannt D-Scoop, das Resultat dieses Vorhabens, wird formal beschrieben mit einer Transaktionssemantik, welche als Spezifikation für Implementationen verwendet werden kann. Ein Prototyp von D-Scoop zeigt, dass Interferenzfreiheit ohne Einbussen von Effizienz erreicht werden kann.

Für letzteres präsentiert diese Arbeit gleich zwei Techniken: *Slicing*, eine Technik um Berechnungen über Felder zu parallelisieren; und *Immutable Classes*, eine Technik um komplexe Datenstrukturen auf Unveränderlichkeit zu überprüfen.

Mit unserer Arbeit an der Integration von gemeinsamen Speicher und sicherem verteilten Rechnen können wir lokale Nebenläufigkeit und Parallelität mit verteilten Systemen kombinieren um ein generelles Modell zu formen.

Chapter 1

Introduction

Concurrency is an important part of modern application and system development. It is a natural result of the capabilities and purposes of devices, and their need to communicate with each other. Depending on the situation, concurrency presents itself in various forms, which includes parallelism and distribution. In the past, frameworks and libraries focused on handling a particular form of concurrency while ignoring the others. But regardless of the form, the problems an pitfalls are similar, and in many systems all forms of concurrency coexist. With specialized approaches for the different forms of concurrency, the same problems need to be solved over and over again. It is therefore important to look for an approach that can reasonably handle all forms of concurrency to ease development and reduce the complexity of systems.

1.1 Background

Distributed programming. Inter-device communication is becoming ubiquitous, and the number of connected devices is growing every day. With this ubiquity comes an increasing demand for programmers to be able to write reliable distributed software, yet this is no simple task.

Various language abstractions have been proposed to make it easier to write distributed programs. One such abstraction, natural for the object-oriented paradigm, is *network objects* [6]: objects whose methods can be invoked over a network. By handling communication in method calls, network objects allow for local and remote objects to be treated uniformly, without regard to their physical location. In principle an elegant generalization; in practice, languages supporting them are often lightweight on synchronization, leaving the user to manage it explicitly, and potentially exposing them to synchronization errors such as data races.

Multi-threaded programming. Writing a multi-threaded program can have a variety of very different motivations [54]. Oftentimes, multi-threading is a functional requirement: it enables applications to remain responsive to input, for example when using a graphical user interface. Furthermore, it is also an effective program structuring technique that makes it possible to handle nondeterministic events in a modular way; developers take advantage of this fact when designing reactive and event-based systems. In all these cases, multi-threading is said to provide *concurrency*. In contrast to this, the multi-core revolution has accentuated the use of multi-threading for im-

proving performance when executing programs on a multi-core machine. In this case, multi-threading is said to provide *parallelism*.

Programming models for multi-threaded programming generally support either concurrency or parallelism. For example, the actor model [1] and various active object-type languages [34] including SCOOP are typical concurrency models: they are optimized for coordination and event handling, and provide safety guarantees such as absence of data races. Models supporting parallelism on the other hand, for example OpenMP [16] or Chapel [12], put the emphasis on providing programming abstractions for efficient shared memory computations, typically without addressing safety concerns.

While a separation of concerns such as this can be very helpful, it is evident that the two worlds of concurrency and parallelism overlap to a large degree. For example, applications designed for concurrency may have computational parts the developer would like to speed up with parallelism. On the other hand, even simple data-parallel programs may suffer from concurrency issues such as data races, atomicity violations, or deadlocks. Hence, models aimed at parallelism could benefit from inheriting some of the safety guarantees commonly ensured by concurrency models.

Since distribution shares many pitfalls with multi-threaded concurrent programming, it is reasonable to look for a common solution. Several languages and libraries attempt to make it easier and safer to write concurrent programs, providing their users with high-level abstractions as diverse as transactional memory [63], block-dispatching [22], actors, and active objects. Given the many shared synchronization challenges, a number of these abstractions have been successfully applied across novel distributed programming approaches, exemplified by languages such as Creol [31], JCoBox [59], and AmbientTalk [17].

1.2 Unification and its Challenges

From all the challenges in the area, the need for a uniform framework that works at the scale of single cores equally well as in vast networks of nodes becomes apparent. This system has to accommodate all the requirements while helping to avoid the critical problems.

The SCOOP programming model is a variant of the active object concept. It was first introduced by Meyer [42] and stands out because it offers an integrated solution to high-level data races. However, it is currently restricted to operate on single machines and does not take advantage of shared memory, and thus cannot compete with the more liberal approaches for parallelism, such as multi-threading, in speed and memory-efficiency as well as distributed models.

In our search for a unified model parallel, concurrent and distributed programming, we focus on the following main challenges.

1.2.1 Safe usage of Shared Memory

While concurrency models such as active objects and SCOOP reduce some concurrency issues, they do so by avoiding the use of shared memory. However, sharing memory helps reduce memory consumption and it is a fast bulk communication medium since it avoids copying. Shared memory could be used safely for parallelizing computations on arrays as well as handling immutable data.

1.2.2 High-Level Data Races

Data races, a form of atomicity violation, together with deadlocks, are the main problems a developer faces when developing a concurrent system. Deadlocks freeze parts of the system, making it easy to find the circular dependencies, which, in comparison to data races, is the simpler problem to handle. Data races can cause corruptions that become apparent at a time where the context in which the data race occurred is already lost, making them notoriously difficult to debug.

Modern approaches such as active objects eliminate low-level data races by elimination of shared memory. However, this is only sufficient in simple interactions; With more complex interactions, data races are again a problem.

1.2.3 Failure

We expect that parts of a distributed system can fail without compromising the whole. When this happens with active objects, a client is informed through exceptions or other forms of errors, while a supplier has little to rely on for recovery, since it is missing the context. In classic message passing systems, this is rarely a problem since the supplier follows a strict process and is informed of the client's disappearance. Object-oriented approaches such as active objects forgo this to achieve greater flexibility.

1.3 Hypotheses and Contributions

Our goal is a system that unifies distribution, concurrency and parallelism with a focus on the challenges above. We start by taking a concurrency model, SCOOP, with a promising approach for handling high-level data races, and extend it to also overcome the other challenges. Therefore, our hypotheses are:

1. An object-oriented concurrency model based on message passing can be extended to a programming model for distributed systems with supplier-aware failure mitigation while maintaining a competitive efficiency
2. An object-oriented concurrency model based on message passing and shared memory can be safely combined without complexity through annotations

We address hypothesis 1 by presenting an extension to SCOOP for distribution with a novel approach to failure mitigation that integrates the supplier.

We address hypothesis 2 by presenting two simple techniques: *slicing* and *immutable classes*. These techniques can be applied to a wide range of object-oriented concurrency models.

1.3.1 Contributions

We provide a programming model with interference-free and transaction-like reasoning for distributed objects, and a runtime that efficiently handles the synchronization including formal specification of the runtime behavior and the protocol.

We provide a set of terms that can be used to describe these as well as a comparison of current active object models.

We present two techniques for safe, efficient and simple usage of shared resources:

First, we handle handle data-parallelism on arrays using a specialized API and a technique called *slicing*.

Second, we handle shared immutable data through *immutable classes* by showing how to prove this property without further annotations.

We provide extensions to make the model suitable for parallel and distributed computing, including formal semantics for the distributed protocol as a guideline and specification for implementations.

1.4 Plan of the Thesis

The thesis can be divided into three parts. The first part is attributed to the introduction of existing work, while the other two parts contain the main contributions.

1.4.1 State of the Art

The first part starts with chapter 2 that defines a set of terms to describe concurrent and distributed object-oriented models and uses it to give an overview of models suitable to fulfill the hypotheses of the thesis.

The Scoop model From the models in the previous chapter we selected Scoop for further investigation. We give an informal introduction to this model in chapter 3, followed by two examples in chapter 4.

1.4.2 A distributed, object-oriented programming model

The second part of the thesis generalizes Scoop for distributed usage. While Scoop is based on the actor model, its unique non-interference guarantee makes an efficient distributed implementation difficult. In addition, we extend the model with transaction-like reasoning for supplier-aware failure mitigation.

The D-Scoop framework. Chapter 5 introduces the distributed Scoop framework. It explains the user-visible changes and the underlying network protocol. A comparison with Java RMI shows that it maintains a competitive performance while maintaining the Scoop guarantees.

The D-Scoop protocol. Chapter 6 gives a formal specification of the D-Scoop system and protocol. The specification comes in the form of an abstract transition semantics, reducing its complexity while detaching it from the programming language used. The semantics gives a deep insight into the workings of the protocol and allows developers to check their implementations.

1.4.3 Taking Advantage of Shared Memory

The third part of the thesis focuses on taking advantages of shared memory while working with a concurrency model that prevents data races. Scoop and most active object models maintain a clear separation of memory regions. We can show that this restriction can be relaxed in various situations without impacting the safety of the model. The results are applicable to Scoop, but also to other concurrent programming models.

Array Slicing. With a technique called *slicing*, presented in chapter 7, we contribute a library-based abstraction that, while being strict in terms of safety as seen and experienced by the programmer, enables the framework to use shared memory where possible.

Immutable Classes. In chapter 8, we introduce a proof framework for immutable classes in Eiffel that does not require annotations while allowing great flexibility in the implementation. Instances of immutable classes can be safely accessed by all processors since they are statically ensured to not be changeable after creation.

Chapter 2

Distributed Object-Oriented Programming Models

Object-oriented distributed programming languages usually either have a distinct notion of processes and communication or use an object-oriented abstraction for one and/or the other. Approaches for such abstractions can be divided into network object and active object approaches. The network object approaches go back to Modula-3 [6] and are still used in widespread applications through middleware such as Java RMI and CORBA. They use a distinct notion for processes, but employ feature calls as an object-oriented abstraction for communication between machines. Although their origin is different, active object approaches share the communication abstraction with network objects but furthermore also use an object-oriented abstraction for processes.

The purpose of this chapter is to give an overview of the various approaches for object-oriented distributed programming with a focus on active objects. This overview is separated into two parts. First, a common set of terms is established, which by themselves present the various forms of handling concurrency and distribution. Second, a comparison of active-object based approaches gives an understanding which combinations have been successfully used.

2.1 Terminology

The terminology used to describe a concurrency model differs from author to author, making it difficult to see commonalities. As a consequence, we introduce a terminology for usage with distributed object-oriented models.

2.1.1 Node

A distributed system is made up of *node*s, which themselves contain regions and processors. Shared memory is generally only available within a node, and nodes communicate with each other using some form of message passing. A node can be a physical machine or some abstraction thereof, for example a virtual machine, a container or a (heavyweight) operating system process. It is also possible that multiple machines form a node, for example to add reliability against hardware failure, but even then, the node is perceived by the other nodes as a single entity.

2.1.2 Network Objects

A *network object*, is an object whose methods can be invoked remotely: over a network. The abstraction is a simple but natural generalization of standard objects to distributed contexts: the programmer interacts with their interfaces in the same way as usual, without regard to where the objects are located. Communication is handled through feature calls: the node where the object is located, the supplier node, is executing the call, while the client node either stalls or continues execution.

The term `network object` first appeared in context of Modula-3 [6], and have since strongly influenced Java's Remote Method Invocation (RMI) API as well as the Common Object Request Broker Architecture (CORBA) standard. However, earlier systems such as Argus [37], Emerald [7] and distributed Smalltalk [3] already introduced similar abstractions. Note that these models only support synchronous calls, to mimic normal non-distributed calls, but we use network object as a general term to refer to any objects whose features are available for remote calls, regardless of the semantics. Therefore, our definition of network object extends to objects whose features can be called asynchronously, such as active objects in models we present in section 2.4.3.

2.1.3 Processor

There are various terms for describing the engine that executes code, for example: processor, process, thread, actor or even just active object. For easier comparison, we use the term *processor* as an abstraction for any form of execution engine, even if the model does not separate the executing engine from the object(s) it is responsible for: a processor, in our terminology, is not an object, but in some models, (active) objects and processors have a one to one relationship or some objects are used to influence the processors' behavior. We chose this term because process, thread and actor all imply a specific concurrency model. A processor may have some transient state such as a local variables stack, message queues and an instruction pointer, which are essential for its operation, but not the application logic or data. The term processor was coined originally in relation to Scoop [42]. A concurrent or distributed system consists of a variable number of processors running concurrently.

2.1.4 Region

A *region* is a set of objects. In most cases, regions are not overlapping, that is, an object belongs to exactly one region. Objects belonging to different regions are considered *separate* to each other.

The processors that are allowed to access objects in a region can be restricted. *Shared memory* is a region that is shared by multiple processors, that is, can be accessed by all of them concurrently.

The access policy of regions is a core defining criteria for concurrency models. For example, multi-threading provides a large shared region that can be accessed by all processors (threads) of a node, while active objects allow just one processor per region to avoid low-level data races.

2.1.5 Features, Calls and Application

We use *feature* as a collective term for functions, procedures and attributes of an object or class. Functions and procedures are also referred to as routines.

Feature call

A feature call is the request for application of a feature to a specific object. The calling and the called processors may be different, in which case it is a *separate call*. A *query call* yields a result, so it is a request to apply a function or read an attribute, while a *command calls* is to apply a procedure or write to an attribute. Some models refer to this as sending a *message* to another process or object.

Feature application

We refer to the *feature application* as the actual execution of a feature due to a *feature call*. The application is performed by the called processor, which may be different from the calling processor. We use the term *non-separate* if it is done by the same processor, and *separate* for the other case, similar to the usage in Scoop [42].

2.1.6 Client / Supplier

A *client* is the object, processor or routine that issues a features call, while the *supplier* is the recipient of the call. We use the more specific terms client/supplier object, client/supplier processor or client/supplier routine where necessary.

2.1.7 Communication

There are two basic models for communication: message passing and shared memory.

Message passing means that processors are sending and receiving messages. A client may wait for a supplier to read the message or even for the supplier to send an answer. At the same time, a supplier may wait for a specific message to arrive. Message passing can therefore be synchronous or asynchronous depending on the specific concurrency model and the protocol. A more specific variation in object-oriented models such as network objects and active objects uses messages to pass separate feature calls, with many different semantics as detailed in section 2.2.

With the right data structures, shared memory, that is, regions accessible by multiple processors, can be used as a medium for communication within nodes. However, leveraging shared memory in such a way requires synchronization primitives such as locks. Using shared memory as a communication medium has two advantages: first, it is flexible regarding the form of communication. For example, message passing can be realized with shared memory by using a queue and a monitor. Second, with shared memory it is possible to send large amounts of data without the need to copy the data to reduce memory requirements and increase efficiency.

2.2 Concurrency

While there are many more variants of communication and sources of concurrency with message passing and especially shared memory, we give specific definitions for those relevant for object-oriented concurrency models as these are the focus of this chapter.

2.2.1 Asynchronous Calls

A major source of concurrency in some object-oriented concurrency models is the ability to make asynchronous calls. In this case, the client does not wait for the application of a feature to finish before continuing, but it may wait until the call was received.

2.2.2 Synchronous Calls

For various reasons, not all calls can be asynchronous, so some models allow for synchronous calls. In this case, the client waits until the supplier finished the application of a feature. This can be used as a form of synchronization to make sure all the previous asynchronous calls have been executed if the model ensures that calls are handled in the order they have been issued. It may also be a solution if the call yields a result needed by the client.

2.2.3 Future Calls

Instead of making all calls that yield a result synchronous, it is also possible to use the concept of futures [67]. In this case, the call is asynchronous but instantly yields a result, the *future* or *promise*. The future is a placeholder for the actual result. The client can defer blocking until the result is actually needed. A future can also be used with command calls as a way of synchronization, in which case the client can use the future to wait until the associated feature application is finished.

2.2.4 Asynchronous Replies

A third option for calls yielding a result is for the supplier, instead of simply returning the result, to issue an asynchronous callback to the client. When making a separate call, the client can declare a continuation or closure to be run when the callback arrives. This allows the client to make the call asynchronous and completely finish its execution without waiting. A concurrency model that integrates asynchronous replies therefore allows for complete asynchronous execution which in turn avoids deadlocks by design.

While almost every model allows for this type of callback in some manual way, only some models integrate it to give the programmer the ability to request callbacks instead of replies regardless of whether the supplier is prepared for it or not. A variation of this allows for sending the result as an argument of a call to an object different from the client of the call.

2.2.5 Local/Remote Wait Condition

Remote wait condition allow clients to await state changes in a supplier before continuing execution. For example, a consumer in the classic producer/consumer example may await for the shared buffer to not be empty. While it is possible to implement wait condition using other forms of synchronization, some languages have explicit support. The term *general wait condition* can be used if wait conditions are normal programming language expressions without restrictions.

Another form of wait conditions are local wait conditions. In this case, the processor suspends the current call and starts processing further calls. When the condition

is satisfied, the suspended call is continued. This implies that calls are not handled in a strict order.

2.3 Principal Problems in Concurrency and Distribution

While there are many different possible problems associated with concurrency and distribution, this work focuses on two: high-level data races and failure.

2.3.1 High-Level Data Races

The occurrence of a data race is typically defined, e.g. in [58], as follows: a *data race* occurs when two processes access a shared resource and when

- at least one access is a write, and
- the processes use no explicit mechanism to prevent the accesses from being simultaneous.

Many concurrency models do not allow shared memory or other shared resources, which is often seen as a prerequisite for data races due to the definition above. We use the term *low-level data race* for these data races. However, a form of higher level atomicity violations can also best be described as a data race. For example: Assume we have a network object of the bank account class in listing 2.1. Not that although the listing uses Eiffel, it is not Scoop. We assume that only a single call can be executed at a time, that is, the object acts as a monitor [29].

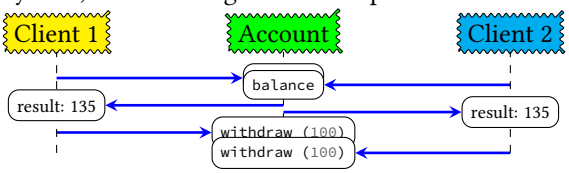
```
class
    BANK_ACCOUNT

feature
    balance: INTEGER

    withdraw (a_amount: INTEGER)
        require
            a_amount <= balance
            a_amount > 0
        do
            balance := balance - a_amount
        end

    deposit (a_amount: INTEGER)
        require
            a_amount > 0
        do
            balance := balance + a_amount
        end
end
```

Listing 2.1: Bank account example

Now assume that a client issues the following instructions:

```
if account.balance > 100 then
    account.withdraw (100)
end
```

We can see, that the intention is to withdraw something on the bank account. However, these instructions cause two separate requests. With other processors in the system, the following situation is possible:



This is very similar to a classic data race as the consequences are the same. In a way, the request queue of the bank account is a shared resource with command calls as *write* instructions and query calls are *read* instructions. With these definitions, the situation above is correctly identified as a data race, albeit a *high-level* one.

One might note that this data race can be avoided by modifying `withdraw` so that it does not change the balance if the amount is higher. However, it might not always be possible to easily adapt a supplier to the needs of a client.

High-level data races are often called *atomicity violations*. We use this term to refer to both high and low-level data races.

2.3.2 Failure

A distributed system should be robust enough to keep on working even if some of the nodes vanish. To achieve this, the program has to be able to mitigate the failure. Mitigation in the client can simply be done by handling the appropriate exception. Handling the failure in the supplier, however, is often difficult.

For example, we assume two a bank accounts `a` and `b` like in listing 2.1 and a client executing the following code:

```
if (a.balance > 100) then
    a.withdraw (100)
    b.deposit (100)
end
```

Apart from the data race, there is another problem that only shows up in a distributed context. Suppose that the client resides on a different machine, and has already issued the `withdraw` call. Let us look at two possible scenarios.

In the first scenario, the machine where `b` resides crashes. The client is informed of the problem by an exception and `deposits` the money back onto `a`.

In the second scenario, the client crashes, so the money is lost. This calls for a mechanism that enables the supplier to mitigate failure if a client fails.

2.4 Concurrent Programming Paradigms

Below we give a short overview of the general paradigms for object-oriented concurrent and distributed programming models.

2.4.1 Threads – Classic Shared Memory

Threads are processors with a small exclusive region containing the thread local data, and a big shared region where most objects live. Since these processors are able to access all object in the shared region, synchronization measures such as locks are needed to avoid data races. Some of these measures, such as signals with monitors, have a message-passing semantics. However, shared memory models are usually not bound to a specific communication medium. Most programming languages used in industry support threads. Threads are a thin abstraction over the underlying architecture and as such a good basis to build more elaborate models upon.

The threading model can be combined with network objects to form a distributed system. The framework usually uses proxy processors to mimic the client in the supplier node, and objects need to be exported before they are reachable from the network. For transparency, separate calls are synchronous.

2.4.2 Processes – Classic Message Passing

Processes are a combination of processor plus an exclusive region. Communication between the processes is realized through message passing. Processors may choose to block until receiving a message matching a certain pattern, which allows synchronous and asynchronous communication. Process calculi such as CSP [30] are examples, as is the actor model [1] and many protocols such as IMAP [15].

This form of communication and concurrency requires strict specification on how interactions between individual components are handled, which can be done by using process calculi. This also results in less flexibility: every communication follows a strict protocol. Furthermore, it treats communication completely orthogonal to the object-oriented abstraction. Nevertheless, it is the most general form of distributed communication and the basis for other distributed models. Note that network objects, although it follows the shared memory paradigm, is usually implemented over message passing.

2.4.3 Active Objects

The concept of active objects is an application of the actor model [1] to the object oriented paradigm. The idea behind active objects is to use objects for transparently representing actors, and use feature calls for communication.

In classic active object models, regions contain a single *active* object and processors have access to exactly one region. Communication usually allows for synchronous, asynchronous and future calls. In this case, the active object represents itself, the region and the processor at the same time. Some recent models allow for multiple active objects per region. This is more flexible, but requires the programmer to differentiate between the object and the region.

Active object models are generally simple to apply to distributed systems their communication medium is already based on messages.

Passive objects. A *passive* object, as opposed to an *active* object, cannot be referenced from another region. As a consequence, separate calls cannot be targeted at passive objects. References to passive objects in arguments of separate calls lead to a deep copy[1] of the passive object: the object and all passive objects it directly or

[1] A deep copy contains a copy of all the directly or indirectly referenced passive objects.

indirectly references are copied, but not the referenced active objects.

Some models, for example by using the active object design pattern [35] in a shared memory environment, put passive objects in a shared region, where all processors can access them, allowing for parallel access but requiring synchronization to avoid data races.

2.5 Comparison of Network Object Languages

The number of languages intended for distributed computing is huge; an early attempt at giving an overview over distributed programming languages in [2] from 1989 already listed 92 languages. Since this work focused on object-oriented distribution, we only consider models that make use of network objects. While we acknowledge that there are good actor-based languages, such as Erlang and Scala/Akka providing good fault tolerance, we are not including them in the comparison, as they are examples of classic process + message-passing communication, not network objects.

In this area, we can make a distinction between pure network object models, which leave synchronization to the user, and active object models. In this section, we look at the former models and refer for active object-based languages to section 2.6.

Many modern object-oriented languages offer a method to use objects remotely. In general, these concurrency models have one shared region per node that is accessible for all processors. Every remote call, usually synchronous, is executed by another processor on an object in the shared region. This means that multiple executions with the same target can run in parallel if there is no manual synchronization employed. For our comparison we chose a few important languages:

Modula-2 The language that coined the term *network object* and introduced the garbage collection method still used by most network object languages.

Emerald A language best known for its ability to move objects.

Argus A language that integrates transactional and data integrity features of database systems with a network object architecture.

Java RMI Remote execution support built into Java that also allows the transfer of bytecode.

CORBA The standard for network objects across different programming languages.

2.5.1 Evaluation

For the evaluation of the network object languages, we focused on the following properties:

1. What communication modes (synchronous, asynchronous etc.) does the language support?
2. How does it ensure data integrity on failure, especially incomplete computations?
3. How does the system recover from failure?
4. What are the mechanisms for synchronization?

2.5.2 Argus

Argus [37] is a programming language built for reliability.

Argus objects are relatively heavy-weight and form regions. Every region has a variable number of processors associated with it. These processors are only associated with one region and can only access objects located there. Every region has a *guardian* object that offers the features accessible from other regions. Objects in a region can be *stable*, which means that their state is backed by nonvolatile memory, so that the region can be recovered in case of a failure.

Argus supports transactions, that is, computations that are serializable and total. Transactions are implemented using special *atomic objects* that provide the needed support for synchronization and rollback of changes. Synchronization is pessimistic using reader-writer locks.

Communication modes. Synchronous only.

Data integrity. Argus uses atomic objects to ensure totality: actions consiting of multiple computations are either applied atomically in their entirety or not at all.

Failure Handling. Non-transient objects are synchronized with stable memory to allow for automatic recovery. Recovery routines are automatically started when a node needs to be restarted due to failure.

Synchronization / Data races. The transaction-like support in Argus can be used to prevent data races.

2.5.3 Emerald

Emerald [] is a programming language thats support mobile objects and processes.

In Emerald, all object of a node are placed in a shared region, accessible by all processes. So-called *active objects*, not to be confused with the term introduced by us before, contain a process that is started when the objects has been created. However, this process does not have exclusive access to its object, but it may invoke private features that are not accessible by other objects. If the object is declared as a monitor, all operations on the object are mutually exclusive.

Emerald supports mobile objects and processes: language constructs allow objects to be moved and fixed to nodes. If an object with an associated process is moved, the process is moved too.

Communication modes. Only synchronous communication.

Data integrity. None.

Failure Handling. No.

Synchronization / Data races. Emerald supports monitors together with condition variables. However, it is not possible for a client to explicitly acquire the lock, as with the original monitor concept, feature calls enter and leave the monitor automatically similar to synchronized methods in Java.

2.5.4 Modula 3

Modula 3 [6] shaped the way how network objects are handled by many distributed frameworks, not through more features but rather less. It only supports synchronous calls and the network object implementation is in general very lightweight. The garbage collection algorithm used by network objects of Modula 3 [5] is still the most widely used one. Most current implementation of network objects, including Java's RMI, CORBA and .NET remoting are all inspired by the approach of Modula 3.

Communication modes. Synchronous only.

Data integrity. None.

Failure Handling. No.

Synchronization / Data races. Only normal programming language constructs.

2.5.5 CORBA

The Common Object Request Broker Architecture [52] is a standard for inter-operable network objects. The required core of the standard is limited, but contains many optional parts and allows the system to be expendable. Since the available features depend heavily on the implementation of the Object Request Broker and the service in general, making absolute statements about CORBA's features and limitations is difficult. The main criticism of CORBA is its complexity [25], which lead to its declining use outside of the embedded programming area.

Communication modes. Synchronous communication, but there exists an extension for asynchronous calls and replies [53].

Data integrity / Failure Handling. Optional: Depends on the specific Object Request Broker implementation, a specification for fault tolerant CORBA exists [51].

Synchronization / Data races. It is possible to let only the main thread perform the work, which then works similar to active objects. Otherwise the various constructs of the programming language used by the supplier are available, but since they are not meant for distributed use, they can usually not be used remotely.

2.5.6 Java RMI

While Java Remote Method Invocation is similar to CORBA as its concept, it is far simpler designed. Current versions of RMI can use the same protocol as CORBA for interoperability. A major advantage of RMI is its ability to transfer objects including their implementation in form of bytecode. RMI also supports *object activation*, which allows for network objects to be created and destroyed on an on-demand basis, that is, only if a client actually has an active reference to the object.

Communication modes. Java RMI only supports synchronous communication.

Data integrity. None

Failure Handling. Limited (Object Activation, only for crashed JVMs)

Synchronization / Data races. Monitors and other programming language constructs. Not possible to acquire locks remotely.
Source: Java SE 8 RMI Specification

2.6 Comparison of Active Object Languages

Although active object languages can be seen as a specific subtype of network object languages, not all active object languages are distributed and would thus qualify. Nevertheless, they share the abstraction of message passing by feature calls. Their main advantages over other, usually more lightweight, network object languages is the absence of low-level data races due to the separation of regions within a node and that they use feature calls also as an abstraction for synchronization. However, this advantage comes at the cost of having no access shared memory, which places an overhead on many parallel algorithms.

As with the network object languages, we picked a few interesting and/or widely used languages for a comparison. The presented languages together give a overview of the current state of the art, even if they are sometimes decades old: we chose the well-known pioneers. Not all these languages support distribution, but this does not make them irrelevant. The selection is the following:

ABCL A set of languages that build upon the actor model and can be seen as the predecessor to the modern active object approaches since it pioneered many modes of communication.
ASP A calculus for active objects built upon ABCL.
MultiASP An extension to ASP to incorporate threading within an active object.
Creol A classic active object-language used as a basis for many extensions.
E The E programming language is unique due to is complete asynchronous, non-blocking concept that avoids deadlocks and the sharing of processors.
AmbientTalk A language that builds upon the principles of E and extends them to better support ad-hoc networks.
(J)CoBox An active object language that allows referable objects per region and the nesting of regions.
Scoop A concurrency model with a particular focus on avoidance of data races.

2.6.1 Evaluation

For every model, a short introduction using the terms above is given before the evaluation. The evaluation consists of the following items:

Shared Processors. Can objects share their processor?

Synchronization. What modes of synchronization are available at the language level?

Data races. Does the model avoid low-level data races? Is it providing the programmer with means to avoid high level data races? Is it reordering calls, therefore making it difficult for the client to rely on a specific state?

Shared memory. How can a programmer take advantage of available shared memory to increase performance without risking low-level data races?

Callbacks. How does the model handle callbacks avoid a deadlock? Is it important for the client to know whether or not the supplier will send a callback?

Interruptions. How can an active process be interrupted, for example, to cancel an operation?

2.6.2 Failure

While AmbientTalk tries to avoid failure due to intermittent disconnections, none of the programming models offered an advanced mitigation strategy for failure. While not an active-object language and therefore not part of this comparison, Argus [38] offers atomic objects for this purpose.

2.6.3 Overview

Table 2.1 gives a brief overview on the findings of the evaluation.

The *Synchronization* column shows the different types of feature calls and synchronization methods possible. *A* and *S* are for basic synchronous and asynchronous feature calls, while *F* and *R* indicate future calls and asynchronous replies. *W* stands for remote and *L* for local wait conditions.

Data Races can be handled either basically, which just prevents low-level data races without making it possible to make multiple calls that are all considered one operation. *Reordering* indicates that message calls not need to be handled in the same order as they arrived. *Exclusive* states that the languages supports clients making multiple calls without interruptions. *No* means that the same pitfalls as with multithreading are also possible in the model.

Callbacks are either *supported* in general or realized by *suspending* the current execution in favor of other calls until the message with the result arrives. A fourth option, *explicit*, indicates that the client has to anticipate the exact callback message, which is not flexible and basically amounts to no support for callbacks.

Model	Shared Proc.	Synchronization	Data Races	Shared Memory	Callbacks	Interrupt
ABCL	No	A,S,F,(R)	Basic	No	Explicit	Yes
ASP	No	A,(S),F	Basic	No	Explicit	No
MultiASP	No[a]	A,(S),F	No	Yes	Explicit	No
Creol	No	A,S,F,L	Reordering	No	Suspending	No
(J)CoBox	Yes	A,S,F,L	Reordering	No[b]	Suspending	No
E	Yes	A,R	Basic	No	Supported	No
AmbientTalk	Yes	A, R	Basic	No	Supported	No
Scoop	Yes	A,S,(F),W	Exclusive	No	Supported	No[c]

[a] The authors remark that it is possible to extend the calculus accordingly
[b] JCoBox's compatibility with Java allows usage of libraries that are based on threads
[c] A mechanism called *Duels* [50] is proposed to allow interruptions, but has not surfaced yet

Table 2.1: Overview of capabilities of selected concurrent and distributed object oriented programming models

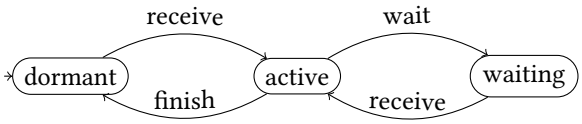


Figure 2.1: Processor states in ABCL/1

2.6.4 Actor Based Concurrent Language

ABCL is a group of object-oriented concurrent programming languages. Lisp-based ABCL/1 [67] combines many many aspects that are now common in active object programming models, like asynchronous and future calls.

Object model

Every object in ABCL forms their own region, each with one processor. In ABCL/R, the meta-objects serve as the processors for the objects they denote. ABCL/R2 introduces meta-object groups to allow for more cooperation between the meta-objects introduced in ABCL/R. Objects in ABCL, with exception of ABCL/C+, are not type checked, which in the case of ABCL/R and ABCL/R2 allows objects to change their features at runtime through their meta-objects. Note that due to ABCL (except ABCL/C+) being based on LISP, data structures other than objects are possible as values.

Processor state

Initially, a processor is *dormant*. If it receives a feature call in one of its queues, it changes to *active*, where it stays until the request if finished, in which case it goes back to *dormant*, or waits, changing the state to *waiting*. The three states and their changes are shown in fig. 2.1. Later active object languages often use similar states for active objects.

Processing model

Every processor has two message queues for incoming calls: an ordinary and an express queue. If a call is received in the express queue while the processor is *active*, it suspends the current operations and processes the express call before either continuing with the ordinary call or aborting it. Messages originating from one processor are received in the order they are sent. We do not qualify express messages as reordering since the purpose of express messages are to interrupt a processor. In the well known producer-consumer example, they could be used to stop a producer. As such, they can be seen as a tool similar to exceptions.

Evaluation

Shared Processors. Objects have their own exclusive processor.

Synchronization. ABCL allows for asynchronous, synchronous and future calls. A call can continue after the result is sent back. In addition, a call may await incoming calls meeting a specified constraint. For example, a shared buffer's `get` feature may await the call of the `put` feature if the internal buffer is empty. ABCL/C+ [19]

introduced the possibility to specify a recipient for the result of asynchronous calls, mimicking asynchronous replies.

Data Races. There is no integrated approach for a client to ensure that its subsequent calls are not interrupted by other processors. No shared memory implies safety against low-level data races.

Shared Memory. ABCL does not support shared memory.

Callbacks. A client can explicitly await incoming calls that match a specified pattern, which, among other things, allows for specified callbacks from the supplier and even incoming calls from other clients. The client needs to know exactly whether the supplier will send a callback, it cannot simply wait just in case there might be a callback. Callbacks during synchronous calls lead to deadlocks.

Interruptions. ABCL handles interruptions by separating *ordinary* from *express* calls. The latter are handled by a separate queue and can interrupt the former. If an express call arrives while an ordinary call is applied, the ordinary call is suspended and the express call applied instead. After the express call(s) have finished, the ordinary call is either continued or aborted. The programmer has to be careful to avoid low-level data races between ordinary and express calls.

2.6.5 Asynchronous Sequential Processes

The ASP calculus [10] is a calculus for active objects.

Object model

Objects are divided into active and passive objects. While active objects can be referenced from everywhere, passive object can only be referenced from objects within the same region (called activity). Future references can be passed as arguments.

Processing model

Every region has its own processor and a single call queue. A call can yield execution to later calls matching a specified pattern. For example, a queue object could yield a call to *dequeue* to a call to *enqueue* if the queue is currently empty. After a matching call is processed, the original call continues execution.

Evaluation

Shared Processors. Regions have a single processor, but there is only one active object in the region, all other objects are passive and cannot be referenced from outside their region.

Synchronization. Asynchronous and future calls; the latter can be used to emulate synchronous calls.

Data Races. There is no integrated approach for a client to ensure that its subsequent calls are not interrupted by other processors. No shared memory implies safety against basic data races.

Shared Memory. Not considered.

Callbacks. There is no direct support for callbacks, but a call can yield at some point after the call for receiving a callback. However, since this mechanism blocks until it receives the specified call pattern, a client has to know in advance whether a supplier will send a callback.

Interruptions. ASP does not allow interruptions.

2.6.6 MultiASP

MultiASP [26] extends ASP to allow handling multiple requests in parallel. This is achieved by putting features in groups. Two features in the same group cannot be processed in parallel.

Processing model

Every region has an infinite number of processors. Processors eagerly process calls from the calls queue as long as they belong to different groups than the ones still being computed.

Evaluation

Shared Processors. As in ASP.

Synchronization. As in ASP.

Data Races. Due to the possibility of executing multiple calls in parallel, low-level data races can occur as in multi-threaded applications. The programmer has to choose the correct groups for the features or correctly use synchronization measures such as locks.

Shared Memory. Parallel processing takes advantage of shared memory at the cost of annotations and/or explicit synchronization using locks.

Callbacks. As in ASP.

Interruptions. As in ASP.

2.6.7 Creol

Creol [32] is an object-oriented, distributed programming model designed with type safety in mind, introducing a form of contract that not only specifies the behavior of the supplier, but also of the client.

Object model

All objects in Creol form their own region similar to ASP. *Interfaces* specify the behavior of an object as a set of contracts. A contract specifies the supplier feature, but can also require that the client adheres to a given interface, which can be used for callbacks even if the client does not appear as an argument. While this is similar to selective export of features in Eiffel, the availability of the client in the form of a special variable is a new concept.

Contracts in Creol are not to be confused with contracts as in Design-by-Contract: They only specify the argument types, not arbitrary pre- and postconditions. Interfaces can inherit from multiple other interfaces and subsume their behavior.

Like in most object-oriented programming languages, objects are defined by classes that implement interfaces and can also inherit from other classes. Inheriting from a class, however, does not imply adherence to the same interfaces. Variables in Creol are typed based on the interface of the object.

Processor state

Processors in Creol are either active or inactive. A processor is active while it processes a feature call, but feature calls can suspend execution. Any further call will activate the processor again, and if this call is finished, the original continues.

Processing model

Processors have a single incoming call queue, but may reorder calls via so-called *Method overtaking*: a supplier may evaluate feature calls from a client in another order than made by the client.

Evaluation

Shared Processors. Objects have their own exclusive processor.

Synchronization. Creol allows for asynchronous, synchronous and future calls. A client can suspend the current execution until a (local) condition is satisfied, for example, until a future value is available. This allows for all other incoming future calls to be executed, also from other processors than the current supplier.

Data Races. There is no integrated approach for a client to ensure that its subsequent calls are not interrupted by other processors. The possible reordering of requests poses an additional risk. No shared memory implies safety against basic data races.

Shared Memory. Creol does not support shared memory.

Callbacks. A client can suspend execution, which allows from callbacks from asynchronous calls but also all other incoming calls. Callbacks during synchronous calls lead to deadlocks.

Interruptions. Creol does not support interruptions.

2.6.8 (J)CoBox

The goal of CoBoxes [59] is to enable a better structuring of objects in a distributed system. This model can be seen as an evolution of ABCL and Creol, with some concepts taken from E and Scoop.

Object model

Similar to E and Scoop, regions contain multiple objects and every object can be referenced. However, regions can also be nested. Only objects on the topmost, the same and one lower level can be referenced. A region is created when an object of a specifically marked class is instantiated, similar to `separate` classes in early versions of Scoop.

Processing model

Processors have a single incoming call queue. A call can *await* a future instead of retrieving it to yield for waiting calls. Yielding is also possible explicitly to wait for local state changes. It is not possible to restrict the calls to be executed while waiting, so it is possible for later calls to finish before other calls from the same client are resolved.

Evaluation

Shared Processors. Objects are combined into regions that share a processor. Regions can be nested.

Synchronization. CoBoxes supports asynchronous calls, future calls and with the latter also synchronized calls through blocking. Tasks can yield to await local changes.

Data Races. There is no integrated approach for a client to ensure that its subsequent calls are not interrupted by other processors. No shared memory implies safety against basic data races.

Shared Memory. CoBoxes do not take advantage of shared memory.

Callbacks. Only while yielding, which allows all other calls to proceed, not just callbacks.

Interruptions. CoBoxes do not handle interruptions.

2.6.9 E

The E programming language [44] was designed for secure, capabilities based distributed computing. However, a very interesting property is the pure asynchronous nature of the language, preventing all deadlocks.

Object model

All objects in E are active and can share processors. Objects can be differentiated in *near* objects and *far* objects. *Near* objects share the same processor as the current object, while *far* object have a different processor. For security, E employs capabilities to limit what other objects can do.

Processing model

Processors have a single incoming call queue. Synchronous calls are only possible to *near* objects, so cannot be used as a synchronization mechanism. However, a client can register a continuation for when a promise is fulfilled, enabling asynchronous replies.

Evaluation

Shared Processors. *Near* objects share a processor.

Synchronization. E never synchronizes processors. The language allows for continuations when results become available or the supplier has a problem.

Data Races. There is no integrated approach for a client to ensure that its subsequent calls are not interrupted by other processors. No shared memory implies safety against basic data races.

Shared Memory. E does not support shared memory.

Callbacks. Since feature calls in E are always completed without waiting, a supplier can make callbacks that are normally enqueued and can be handled.

Interruptions. E does not support interruptions.

2.6.10 AmbientTalk

AmbientTalk [65] is a programming language intended for ad-hoc distributed networks. Its main focus is dealing with intermittent disconnections. It is based heavily on the principles of the E programming language, but extends it with better support for failure and support for finding ambient resources.

The runtime handles intermittent disconnections transparently: a failure only occurs if the recipient disappears permanently, that is, after a timeout.

Evaluation

AmbientTalk is identical to E in all evaluated aspects.

2.6.11 Scoop

The Simple Concurrent Object Oriented Programming model is discussed in more detail in chapter 3. The main differentiating feature of Scoop in comparison to CoBoxes is its unique control scheme to handle high-level data races and remote wait conditions. This evaluation is based on Scoop, not the distributed variant D-Scoop presented in chapter 5.

Evaluation

Concurrency. SCOOP is an active object model, but multiple objects can share the same processor, as in JCoBox and E.

Synchronization. Calls can be either asynchronous or synchronous. All query calls (that is, yielding a result) are always synchronous; some commands may be synchronous depending on the arguments to support callbacks. In addition to synchronous calls, the model supports remote wait conditions in the form of preconditions. While it does not have the notion of future calls, the common practice encourages that intensive computations are done in procedures whose results are retrieved in query calls. Since procedures are asynchronous and the query blocks until all previous calls have been applied, this renders an integration of future calls redundant. This form of future calls is only possible due to the non-interference guarantees.

Callbacks. Callbacks are possible for synchronous calls; this is achieved by a technique called *lock-passing*.

Interruptions. There are various proposals for interruptions in Scoop, an early proposal is *duels* [50]. As of today, no integrated approach exists.

2.7 Conclusion

While the discussed languages differ on synchronization principles, most of them do not address safe usage of shared memory and their failure model is rather simple. The only model that has a mechanism for shared memory and a more advanced failure model is Argus. However, in Argus, this is limited to the special atomic objects.

We chose Scoop as a basis for improvements in this area since it already offers a solution to high-level data races that includes handling of callbacks.

Chapter 3

SCOOP

The *simple concurrent object-oriented programming* model, SCOOP, comes in the form of an active-object type programming language based on Eiffel [20], but its principles are applicable to other programming languages.

This chapter informally introduces SCOOP using the terminology defined in chapter 2. SCOOP was first introduced by Meyer [42], and a complete description of SCOOP was given in [49], with later formalization by Morandi [47]. However, with the introduction of Queues-of-Queues [66], these semantics became partially obsolete. A comparison between the old and the new, QoQ based semantics, was shown in [13] using graph transformations.

The principal implementation of SCOOP is based on Eiffel, in particular EiffelStudio. We assume basic familiarity with the Eiffel programming language and active-object type languages. A complete tutorial for SCOOP by Bertrand Meyer is available online [4].

3.1 Regions and Processors

Regions in SCOOP can contain multiple objects and is associated with one processor, and every object in the region can be referenced from outside the region. Passive regions, discussed, later, are regions without a processor. Eiffel and therefore SCOOP has a notion of expanded objects, which are treated as values and never referenced.

3.1.1 References

This section discusses properties of references in SCOOP and does not apply to expanded objects and immutable objects (chapter 8); it does, however, apply to slices (chapter 7).

Separate Types

A central aspect of SCOOP is that the type system reflects the locality of referenced objects. The system we use is static and divides references into two groups: non-separate and separate. Regular references are of the former type, while references with the keyword `separate` are of the latter. Non-separate references never point to separate objects, while separate references can point to both separate and non-separate objects.

source / target	**separate**	non-**separate**	**Void**
separate	✓	✓	✓
non-**separate**		✓	✓

Table 3.1: Type conformance for assignment

actual / formal	**separate**	non-**separate**	**Void**
separate	✓	✓	✓
non-**separate**	[1]		✓

Table 3.2: Type conformance for reference argument passing to separate calls

```
ref1: separate C -- May reference separate and non-separate objects
ref2: C -- May only reference non-separate objects
```

Separateness is determined from the perspective of the object that holds the reference, that is, where the particular variable is defined, no matter whether the reference is a local reference or argument of a routine, a result type of a feature or an attribute of an object. This is especially relevant for arguments of feature calls, since a non-separate formal argument of a feature call with a separate supplier is non-separate in regards to the supplier, not the client, so passing a non-separate actual argument is not possible.

Type Conformance

The special value **Void**, which represents a reference to no object, conforms to both separate and non-separate types, but void-safety [43] still applies. References of an expanded or immutable types, as presented in chapter 8, should never be marked separate. For other references, we can distinguish the following cases:

Assignment. Type conformance between separate and non-separate types for assignment is simple and shown in table 3.1. Two types do not conform if the target type is non-separate, that is, not annotated with the **separate** keyword, but the source type is. The more expansive description of the Scoop type system in [49] also describes so-called processor tags. However, the current Scoop reference implementation does not support processor tags which is why we are not introducing them here.

Arguments. Type conformance of arguments is the same as for assignment, but it is checked from the perspective of the supplier object, not the client. This makes a difference if the target of the call is separate. We can translate the type conformance rule to the perspective of the client in a separate call, which results in the rules in table 3.2.

As a client, it is usually not possible to pass any reference[2], with exception for **Void** and immutable objects, for arguments that are non-separate from the perspective of the supplier because the supplier expects a reference to an object living in the same region, which, in the case of separate calls, cannot be statically ensured through the

[2]Expanded objects can be passed, but they are passed by value, not by reference.

result \ target	separate	non-separate
separate	separate	separate
non-separate	separate	non-separate

Table 3.3: Type combinations of separateness

type system without additional annotation such as the aforementioned processor tags. If the compiler is able to infer that a given argument resides on the same processor, it may allow it. However, the current reference implementation does not support this.

Type Combination

Table 3.3 gives an overview of the different cases of target type and result type. If the target of a call is separate and yields a value of a reference type as a result, the type of the reference is always separate, regardless of the written result type. This is due to the fact that if the result type is non-separate, the result points to an object within the same region of the target object, not the client. If the target was separate, then so is the result.

Object Tests

It is possible to use an object test to check at runtime whether a separate expression points to a non-separate object:

```
if attached {MY_CLASS} separate_expression as l_non_separate then
    ...
end
```

3.1.2 Expanded Classes

Expanded classes have a copy-semantics: whenever their instances are assigned to a variable, be it an attribute, an argument, a local variable or a result, they are copied. The copy always resides in the same region as the object that holds the variable. Therefore, variables of an expanded type can never point to a separate object and should consequently not be marked separate. Examples for typical expanded classes are: **INTEGER**, **BOOLEAN** and **POINTER**.

3.1.3 Immutable Classes

Immutable classes are an extension to Eiffel presented in chapter 8. Like expanded classes, their instances are always considered non-separate. But unlike expanded objects, which are passed by value, immutable objects are passed by reference. This is possible because they are statically proven to remain unchanged after creation. Typical candidates for immutable classes are strings, although the standard **STRING** class of Eiffel is mutable. Their main advantage over expanded classes is that they can be referenced and variables of immutable types are polymorphic[3].

[3]Variables of expanded type can only hold instances of exactly this class, not a descendant.

3.1.4 Processor Creation

A new processor can be created by simply making an instance of an object of separate type, for example through a `create` instruction on a variable with separate type:

```
local
    l_var: separate C
do
    create l_var.make
    ...
end
```

Or in the explicit form as in `create {separate C}.make`. Regardless of which form of creation is used, the newly created processor is not controlled (see section 3.2) afterwards.

3.1.5 Passive Regions

Passive regions[4] [46] are regions with no associated processor. When a client processor requests calls to an object in a passive region, it adopts the region temporarily. All calls to the region become effectively non-separate, which makes them synchronous but also avoids some overhead, even though this can also be achieved by some transparent optimizations [66].

Passive regions are a way to reduce the number of operating system threads the program requires and are best suited for mutable data objects. Passive regions borrow the operating system thread of the client for all executions, which implies that all calls to objects in passive regions are synchronous.

3.2 Control and non-interference

An important Scoop concept is *control*. A call targeted at a separate reference is only valid if the separate reference is *controlled*. A reference is controlled if the region it points into is controlled by the current processor, that is, the handler of `Current`. When execution of a feature starts[5], all regions that hold at least one of the arguments of the routine are controlled. More specifically, references are controlled if they are arguments of the routine or if it can be statically inferred that the objects they reference are located within the same region as an argument. The current Scoop implementation in EiffelStudio is able to detect this for expression chains as in for example `x.f(a).g.h(b)`, but this could be extended to also detect whether a local variable is controlled after assignment of a controlled reference to it.

Control is a *static* concept, a static property of a relationship between two processors at a specific point during the execution of a feature. The dynamic counterpart to control is *locking*, discussed in section 3.3: a controlled region is always locked at run-time, while a region that is locked at run-time may not be controlled. Or in other words: the set of *locked* regions is a superset of the controlled regions at the same point during execution.

To give an example for the correct use of control, we go back to the bank account example in the comparison, with an account class as in listing 2.1. We assume

[4] The original term passive processor is ambiguous

[5] This includes the execution of the precondition to check whether it holds.

that a client has a feature for transferring some money from one account to another. The `transfer` feature shown below has three separate calls, `balance`, `withdraw` and `deposit`, which are all valid since their targets, `a_from` and `a_to` are controlled due to being arguments of the routine.

```
class CLIENT
feature
    transfer (a_from, a_to: separate BANK_ACCOUNT; a_amount: NATURAL)
        do
            if a_from.balance >= a_amount then
                a_from.withdraw (a_amount)
                a_to.deposit (a_amount)
            end
        ensure
            a_from.balance >= a_amount implies
              a_from.balance = old a_from.balance - a_amount and
              a_to.balance = old a_to.balance + a_amount
            a_from.balance < a_amount implies
              a_from.balance = old a_from.balance and
              a_to.balance = old a_to.balance
        end
end
```

As a counterexample, the `transfer` routine of the following class is not valid:

```
class BUGGY_CLIENT
feature
    from_a, to_a: separate BANK_ACCOUNT

    transfer (a_amount: NATURAL)
        require
            a_amount > 0
        do
            if from_a.balance >= a_amount then
                from_a.withdraw (a_amount)
                to_a.deposit (a_amount)
            end
        ensure
            from_a.balance >= a_amount implies
              from_a.balance = old from_a.balance - a_amount and
              to_a.balance = old to_a.balance + a_amount
        end
end
```

While control is limiting, it has a major benefit: in between the two calls to a controlled object, and in general to objects of the same region, no interfering call from another processor can be executed. This is what makes the `transfer` feature of our example safe: the balance cannot change in between querying it and withdrawing. Also, we can be sure that the precondition still holds unless the client invoked directly or indirectly a feature that invalidates it.

In addition to arguments, a **`separate`** block can be used to establish control:

```
l_amount := 100
separate account as c_account do
    c_account.withdraw (l_amount)
end
```

The **separate** block uses the same mechanisms as a regular routine to gain control/lock the regions and can be seen as a simple short-hand notation for defining a routine and calling it. However, unlike a routine definition, this construct does not allow for a wait condition. In addition, variables outside of the block can be used inside the block.

Eiffel does not allow the same name for a local variable and a feature of a class. To make sure that there are no conflicts when adding a feature, it is common practice to prepend local variables with `l_` and arguments with `a_`. The latter allows us to name the argument of a setter procedure almost the same as the attribute to be set. We propose a third prefix for controlled variables: `c_`. This is to be used in separate blocks as the controlled alias, that is, after the **as**. With these conventions, it is always obvious which variables are controlled, namely the ones that start with either `a_` or `c_`.

3.3 Locking

To achieve non-interference, the SCOOP system locks the regions of all the objects passed as arguments. However, acquiring a lock does not necessarily mean that some form of semaphore or monitor is used, although this was done in older implementations. In modern implementations of SCOOP, and also in D-SCOOP discussed in chapter 5, it is possible that multiple processors can have a lock on the same region at the same time. In SCOOP, a lock is simply the right to send calls to a region. It is for the runtime system to ensure that the calls are ordered so that no interference can occur. This implies that an issued call might not be executed immediately, it rather could take a long time if the region is still busy with calls from other processors. At the end of a routine, all acquired locks are given back or *unlocked* to inform the system that no more calls are to be expected. Execution of asynchronous calls might still happen well after the client routine finished and gave back its lock.

The runtime system ensures that the calls are ordered in a way that does not allow another client to observe one supplier in the state before and one supplier in the state after execution of all the calls. In other words, it ensures realizability of the whole routine/separate block.

Locked. We say that a region is locked if the current processor has a lock on the region. A reference the current processor holds is locked if it points into a locked region.

3.3.1 Control

We mentioned before that the target of a call needs to be non-separate[6] or *controlled* in order for the call to be valid. All separate arguments of a routine are controlled, as are all expressions for which it can be inferred that their result is within the same region

[6] A processor can be considered to always hold a lock on itself, so non-separate references are trivially controlled.

as a controlled expression. There is a strong relationship between control and locks: a controlled reference is always locked, but not all locked references are controlled. The latter situation can occur due to lock passing or because a reference is not statically ensured to be controlled while actually pointing into a locked region.

3.3.2 Lock Passing

A lock being simply a right for enqueuing calls also means that it can be passed to another processor for the duration of a call. This is done for all synchronous calls to avoid deadlocks: the calling routine might have a lock on a processor the called routine needs. In this case, the called routine might request and receive a new lock on the client, but all calls that are issued with this lock are executed after the calling routine finished. In turn, this might lead to a deadlock due to synchronization.

To avoid this problem, with every synchronous call, separate or non-separate, the locks of the current execution frame are passed on. The called routine now only needs to acquire the locks it needs in addition to the received locks. When a routine finishes, it gives back all locks it received and unlocks the ones it acquired itself.

Synchronous commands. As mentioned before, the SCOOP implementation automatically makes command calls which contain at least one *locked* argument synchronous to avoid, through lock-passing, deadlocks. However, this introduces a new problem of making it difficult and sometimes even impossible to decide statically whether a call is asynchronous or not, which is the reason for the tool presented in [40]. With this analysis tool, a programmer can assign variables with different processors and check whether, under these circumstances, a call is synchronous or asynchronous. The description of SCOOP by Nienaltowski [49] used the controlled instead of the locked state of a reference to determine whether a command call is synchronous, which is a more transparent mechanism than the current.

3.4 Blocking and Waiting

3.4.1 Adaptive Synchronization

Query calls that yield a result are synchronous, as are command calls where at least one of the arguments is locked. So for a call to be asynchronous, the following has to hold:

1. The target object is not in the same region as the client. As mentioned in section 3.1.1, a variable of separate type may reference a non-separate object, so it is possible that the call is actually non-separate at runtime.
2. The call is a command, not a query. If the call yields a result, the client waits until it is available before continuing; this is called *wait-by-necessity*. It is not valid to call a function or attribute like a procedure in Eiffel[7], which makes it impossible to make an asynchronous call to a routine yielding a result.
3. None of the arguments reside in a locked region. If one of the arguments is in a locked region, and the region of the client is also considered to be locked, then the *lock passing* mechanism is used so that the supplier can issue calls, barring the client from using them at the same time, so it has to block. This is discussed in section 3.3.2.

[7]Unless the call is encapsulated by an `agent`, which is similar to a closure.

3.4.2 Wait Conditions

All preconditions (`require` clauses) that involve at least one separate argument are evaluated as wait conditions. This works as follows:

1. The client[8] issues the call
2. The supplier acquires the missing locks
3. The supplier executes the precondition
4. If the precondition holds, then
   (a) The supplier executes the body of the routine
   (b) If the routine is a function, the supplier sends back the result
5. If the precondition does not hold, then
   (a) The supplier releases all acquired locks
   (b) The supplier restarts at 2

The runtime system can optimize when step 5b is applied. The current Scoop implementation only restarts the process if at least one call[9] to one of the supplier was completed.

3.4.3 Attributes as Futures

Scoop does not have a built-in concept for future calls. However, due to its non-interference guarantee, it encourages a split of long running functions into a command that calculates the result and an attribute to hold the result. This way, the client can make an asynchronous command call, continue execution and retrieve the result when needed. With lock passing, this also allows the client to let some other processor retrieve the result.

3.5 Agents

The `agent` contruct of Eiffel is used to wrap features into objects. For example, the expression `agent t.compute (?, 5, var)` creates an object of type `FUNCTION` or `PROCEDURE`, which depends on whether `compute` is a function or procedure respectively. This object can then be used to call the feature. The question mark in the agent expression denotes an open argument, this argument has to be filled when the feature is called through the created agent object. The locality of the agent depends on the locality of the target; if the target is separate, then so is the agent.

Agents for Asynchronous Replies. Agents are a useful method for allowing asynchronous replies: The client first, through a distinct call, gives the supplier a reference to the agent that should be called with the result. It then makes a second call that triggers the computation. Once the supplier has finished the computation, it uses the agent to send back the result. The division into two calls is needed to make sure that the computating call is asynchronous. The `COMPUTER` class in listing 3.1 is an example of a supplier that supports both asynchronous as well as synchronous replies. Note that when using an asynchronous reply, the calls from other processors to the client might get applied before the result: this is the intention behind using asynchronous replies in Scoop.

[8] Here client and supplier denotes the processors, not the objects.
[9] This call has to originate from another client since the waiting processor does not skip calls.

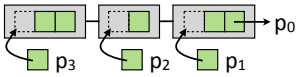


Figure 3.1: Three processors (p_1, p_2, p_3) logging requests on another (p_0)

3.6 SCOOP Runtime

The concurrent programming abstractions presented rely on the existence of a runtime that can correctly and efficiently implement them. At the core of SCOOP's runtime is a simple execution model for managing requests that are sent between processors. Each processor is associated with a "queue of queues" [66], that is, a FIFO queue containing (possibly several) FIFO subqueues for storing incoming requests. Each of these subqueues represents a "private area" for some other processor to log requests, in program text order, and without interference from other processors (since they have their own subqueues). Figure 3.1 visualizes three processors (p_1, p_2, p_3) simultaneously logging requests (green blocks) on another processor (p_0). The processor p_0 is handling the subqueues one-by-one[10] in the order that they were created, and handles the requests within them in the order that they were logged there, hence ensuring the SCOOP reasoning guarantees.

Consider again the processor that calls `transfer (acc1, acc2, 100)` on two separate accounts, `acc1` and `acc2`. Under the current runtime, the handlers of `acc1` and `acc2` both generate a private subqueue on which the calling processor can log requests (i.e. the `balance` queries and `set_balance` commands) without interruption for the duration of the block. Should another processor also need to log requests on an account, then a new private subqueue is generated for it and its requests can be logged without waiting.

We remark that earlier versions of the SCOOP runtime additionally provided timing guarantees by not allowing processors to enqueue requests concurrently [47]. A formal comparison with the current semantics is given in [13].

[10]The client enqueues an unlock request when it no longer needs the queue, that is, unlocks the supplier. The supplier starts with the next queue only after processing this request.

```
class COMPUTER
feature -- Access
    callback: detachable separate ROUTINE[TUPLE[], INTEGER]
    computed_result: INTEGER

feature
    set_callback (a_callback: like callback)
        do
            callback := a_callback
        ensure
            callback = a_callback
        end

    compute (a_arg1, a_arg2: INTEGER)
        do
            ...
            computed_result := (...);
            if attached callback as l_callback then
                separate l_callback as c_callback do
                    c_callback.call (computed_result)
                end
            end
        end
end

class CLIENT
    ...
feature
    my_feature (a_arg1, a_arg2: INTEGER; a_computer: separate COMPUTER)
        local
            l_result: INTEGER
        do
            -- Synchronous reply
            a_computer.compute (a_arg1, a_arg2)
            next_step (a_computer.computed_result)
            -- Asynchronous reply
            a_computer.set_callback (agent next_step)
            a_computer.compute (a_arg1, a_arg2)
        end

    next_step (a_result: INTEGER)
        do    ...    end
end
```

Listing 3.1: Example: using agents for asynchronous callbacks

Chapter 4

Examples

This chapter contains three simple examples for distributed SCOOP. They start with a description and overview of the intended system, followed by the definition of the important classes and their implementation. The following chapters refer to these examples.

We chose the three examples because they represent the three most important basic architectures of distributed systems. The first example, *distributed banking*, is a peer-to-peer system: every bank communicates with the other bank at the same level. The second example, a *chat server*, is a simple client-server model: the clients communicate only with the server, which relays information to the other clients. The third example is a *search engine* that uses worker processors to perform the actual searches before combining them into a sorted result list. Note that in practice, systems often use a combination of these communication principles.

For every example, we give a graphical depiction of a possible state. We use dashed rectangles of various colors to form nodes, squiggly rectangles to form regions and blue ovals for objects. Red arrows are used to denote references.

The examples in this chapter are reduced and simplified for better understanding. Additional features and classes are needed to turn them into full programs. It is possible, due to the seamless integration of D-SCOOP into SCOOP, to understand the basics of the examples by only knowing about SCOOP, the D-SCOOP initialization and compensation mechanisms used in the examples are explained later in chapter 5.

4.1 Distributed Banking

The first example is an extension to the bank account example in chapter 2. It involves bank *agents* that manipulate bank *accounts* in any of the branches of the bank, with each branch handled by a different node.

4.1.1 Architecture

The Distributed Banking system consists of nodes called *branches*. Each branch can hold multiple *agents* which can make transfers between all accounts of the bank[1]. A

[1]Due to the flexibility of D-SCOOP, there is not an actual requirement that agents share the node with the other objects, but in our example they do. Also, these agents are not to be confused with the **agent** construct in Eiffel.

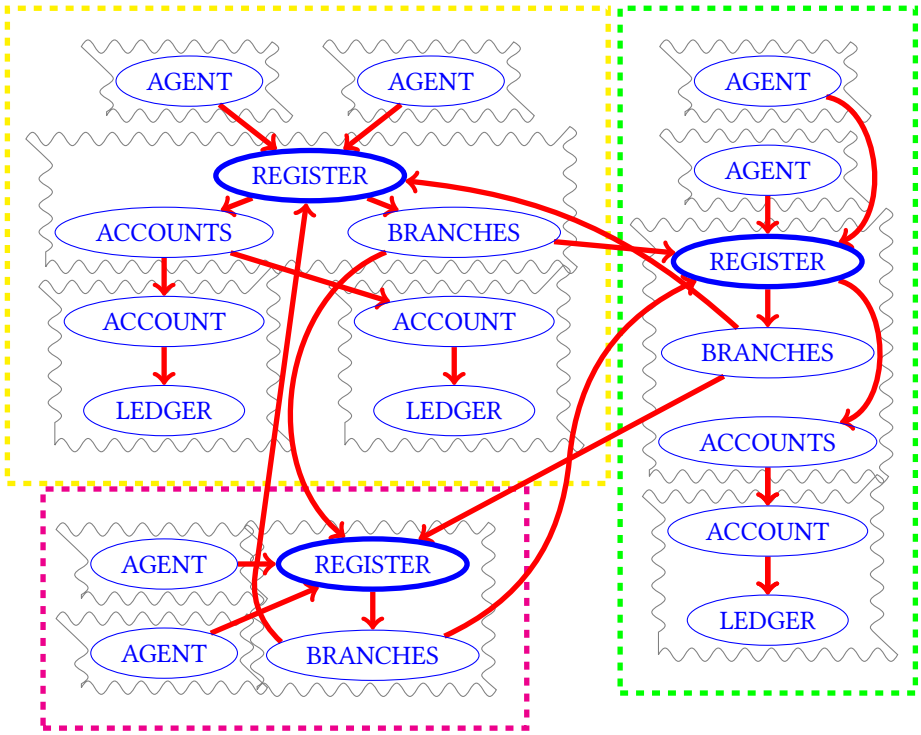


Figure 4.1: Example state of the Distributed Banking system

branch provides an index object called the *register*, which contains a list of *branches* and optionally list of *accounts* where all the accounts of the branch are listed. It is possible that a branch does not manage any accounts, in which case it also does not have an *accounts* object. Examples for those branches are automatic teller machines or small satellite offices.

Every account has an associated list of withdrawals and deposits called the *ledger*. Agents can transfer money from one account to another, which leaves a trace in the ledgers of these accounts.

The agents themselves are suppliers for the different user interfaces, which we chose not to include in this example. Figure 4.1 shows the object structure of a very simple banking system with three branches, where one branch has no accounts, another branch has one account and the third branch has two accounts. Each branch has two agents. The squiggly lines denote the processor boundaries, whereas the dashed rectangles represent the nodes. The index object of each node is highlighted by the bold ellipse.

4.1.2 Initialization

We assume that during startup, a branch receives the address of an available branch to connect to. It can then populate its list of branches from the others, before announcing to the other branches that it is online by sending them a reference to its register.

4.1.3 Classes

Below are the class definitions for our example. We omit the code for setting up accounts and agents.

Account

The account class contains information about the account holder and a reference to the ledger. An account is uniquely identified by the branch plus account identifiers: two different branches may have distinct accounts with the same account identifier.

For the purpose of this example, the account is limited to only contain the identifier and the reference to the ledger. The `undo` feature is only used for compensation (see section 5.4), so it does not provide a compensation for itself. The `compensate`-feature registers the `undo` as a compensation to be run if the client crashes or is disconnected before it unlocked the processor whose region the account object is located.

```
class ACCOUNT

create
    make

feature {NONE} -- Initialization
    make (a_acc_id: NATURAL)
        do
            create ledger
            acc_id := a_acc_id
        ensure
            acc_id = a_acc_id
            balance = 0
        end

feature
    ledger: LEDGER
        -- A list of withdrawals and deposits
    acc_id: NATURAL
        -- The account number

    withdraw (a_amount: INTEGER_64)
            -- Deducts a_amount from the balance
        require
            a_amount > 0
            balance >= a_amount
        do
            ledger.put_front (-a_amount)
            balance := balance - a_amount
            compensate (agent undo)
        ensure
            balance = old balance - a_amount
            ledger.item = -1*a_amount
        end
```

```
    deposit (a_amount: INTEGER_64)
            -- Add a_amount to the balance
        require
            a_amount > 0
        do
            ledger.put_front (a_amount)
            balance := balance + a_amount
            compensate (agent undo)
        ensure
            balance = old balance + a_amount
            ledger.item = a_amount
        end

    balance: INTEGER_64
        -- The current balance of the account

feature {NONE} -- Compensation

    undo
            -- Undo the last withdraw/deposit
        do
            balance := balance - ledger.first
            ledger.start
            ledger.remove
        ensure
            balance = old (balance - ledger.first)
        end

invariant
    balance >= 0
    balance = ledger.balance
end
```

Ledger

The ledger is a list of positive and negative integers, representing deposits and withdrawals. In a real system, it would contain many more pieces of information like the date, a note, the receiving account etc.

```
class
    LEDGER

inherit
    LINKED_LIST[INTEGER_64]

feature
    balance: INTEGER_64
            -- The current balance as calculated from the ledger
        do
            across Current as iter loop
```

```
                Result := Result + iter.item
            end
        end
end
```

Agent

The agent, called `BANKING_AGENT` to not confuse it with the `agent` keyword of Eiffel, can make transfers between two accounts. This is done with the two features `transfer` and `find_account`. A transfer works by first looking up the accounts, then checking the balance of the source account and finally withdrawing the amount from the first account and depositing it to the second account. If a transfer could not be completed, the *success* attribute is set to false, otherwise it is set to true.

```
class
    BANKING_AGENT

feature
    register: separate REGISTER

    success: BOOLEAN

    transfer (
        from_br, from_ac, to_br, to_ac: NATURAL;
        a_amount: INTEGER_64
    )
            -- Transfers a_amount from one account to another
            -- by account id
        do
            success := False
            if
                attached find_account (register, from_br, from_ac)
                    as l_from
            and then
                attached find_account (register, to_br, to_ac)
                    as l_to
            then
                separate l_from as c_from, l_to as c_to do
                    if (c_from.balance >= a_amount) then
                        c_from.withdraw (a_amount)
                        c_to.deposit (a_amount)
                        success := True
                    end
                end
            end
        end

    find_account (a_reg: separate REGISTER; a_br, a_ac: NATURAL):
            detachable separate ACCOUNT
            -- A helper function to retrieve an account from its
```

```
            -- account number and bank branch
        do
            if attached a_reg.find_branch (a_br) as l_br then
                separate l_br as c_br do
                    Result := c_br.find_account (a_ac)
                end
            end
        end
end
```

Register

The register contains references to a list of branches and an optional list of accounts. It provides a facade to these objects for easier retrieval of information. However, a client may also directly use the lists, for example for iteration. The register also contains the branch identifier.

The register also contains the features necessary to connect it to the bank network. For this, the `connect_to` feature has to be called with the address of one bank. The branch then connects to this bank, retrieves a list of connected banks and announces itself to this bank and all the others. This is possible because D-Scoop automatically connects to the nodes of objects it received a reference to.

```
class
    REGISTER

create
    make

feature -- Access
    branch_id: NATURAL
    accounts: detachable ACCOUNTS
    branches: BRANCHES

feature {NONE}
    make
        local
            l_dscoop: DSCOOP
            l_index: detachable separate ANY
        do
            create l_dscoop
            l_dscoop.start_server (agent this_branch, 7000)
            create branches.make
        end

feature -- Branch management
    connect_to (a_address: ISTRING)
            -- Connects the branch to the network
        local
            l_dscoop: DSCOOP
            l_index: detachable separate ANY
```

```
        do
            create l_dscoop
            l_dscoop.connect (a_address, 7000)
            l_index := l_dscoop.last_index_object
            if attached {separate REGISTER} l_index as l_remote_branch then
                branches.extend (l_remote_branch)
                separate l_remote_branch as c_register do
                    across c_register.branches as l_iter loop
                        separate l_iter as c_iter do
                            branches.extend (c_iter.item)
                            separate c_iter.item as c_other_branch do
                                c_other_branch.announce (Current)
                            end
                        end
                    end
                    c_register.announce (Current)
                end
            end
        end

    announce (a_branch: separate REGISTER)
            -- Announces the given branch to the current branch
        do
            branches.add (a_branch)
        end

    this_branch: REGISTER
            -- Simply returns current
        do
            Result := Current
        end

feature
    lookup_branch (a_branch: NATURAL): detachable separate REGISTER
            -- A helper feature to look up a branch
        do
            Result := branches[a_natural]
        end

    lookup_account (a_account: NATURAL): detachable separate ACCOUNT
            -- A helper feature to look up an account
        do
            if attached accounts as l_acc then
                Result := l_acc[a_account]
            end
        end
end
```

Branches and Accounts

These two classes are very similar, both are maps from identifiers to their corresponding references.

```
class
    BRANCHES

inherit
    HASH_TABLE[separate REGISTER, NATURAL]
end

class
    ACCOUNTS

inherit
    HASH_TABLE[separate ACCOUNT, NATURAL]
end
```

4.2 Chat Server

4.2.1 Architecture

The chat system uses a central chat server, running on its own node. Clients, also on their own nodes, can connect to the server, where they get a session object. For the duration of the session, the client holds a lock on the session. This creates a long running transactiona and allows the server to clean up if a client disappears. The server also has a shared list of messages. The clients can retrieve messages and add new messages. By using wait conditions, the client can wait until new messages are available.

The clients are implemented with four processors. Two lists called `input` and `output` and reside in the region of one processor, represent what the user enters and what is displayed on the screen. One processor that is not shown here synchronizes the user interface with these two lists. One processor waits on new input in the input list to send to the server and one processor waits on new messages on the server to send it to the output list.

Aside from being a classic example for a typical architecture, this example also shows that compensations (section 5.4) can be used for long-running transactions. It is possible to use them to make sure some code is run even if the connection was broken or the client is killed. While the example only prints a simple message, a similar mechanism could be used to clean up resources such as file descriptors or database connections.

4.2.2 Initialization

The clients connect to the server and gets its own session object. The session object contains a reference to the shared list of messages. The client does not start a D-Scoop server since no node needs to connect to the client.

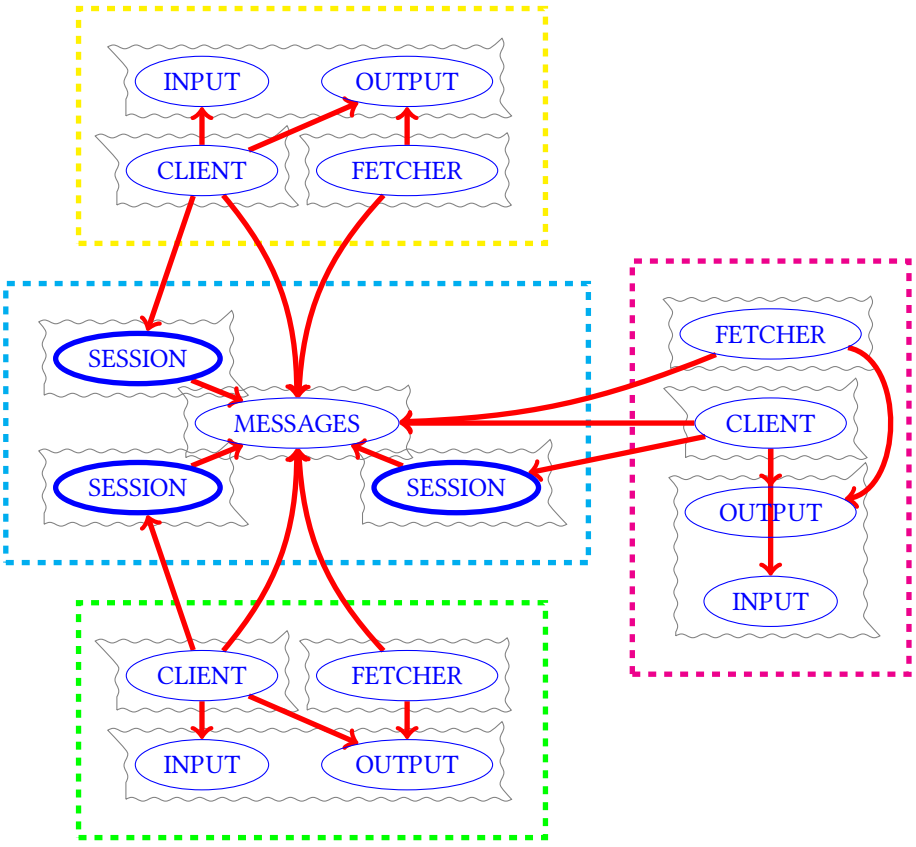


Figure 4.2: Example state of the chat system

4.2.3 Classes

Client

The client is responsible for setting up the session and then accepting the input. It creates the fetcher, which fetches new messages. The client sends a message that the user entered the room. We want the fetcher to get all the messages starting with this message; no more and no less. We can achieve this by acquiring the lock on the messages list, starting the fetcher, which will remember the message count, and then adding the new message. This ensures that no message from another client can slip through.

Wait conditions are used to suspend execution until new input is available. The client saves the last count of items, so that it can detect, through the wait condition, that the count changed and then retrieve the new input. Note that due to the SCOOP semantics, the user interface can enqueue `extend` requests while the client is still retrieving items.

ISTRING is an immutable (see chapter 8) version of **STRING** that automatically converts[2] from and to a mutable string.

```
class
    CLIENT

create
    make

feature {NONE} -- Initialization
    make (a_input, a_output: separate LIST[ISTRING];
                a_server_address: ISTRING)
            -- Initializes the client
        local
            l_dscoop: DSCOOP
            l_index: detachable separate ANY
        do
            create l_dscoop
            l_dscoop.connect (a_server_address, 7000)

            if
                attached {separate SESSION} l_dscoop.last_index_object
            as l_session then
                session := l_session
            end
        end

feature
    session: detachable separate SESSION

    input, output: separate LIST[ISTRING]
        -- These two lists represent input and output of the chat program
```

[2]This conversion can use the **convert** mechanism from Eiffel

```
last_count: INTEGER
    -- The number of lines read from input

username: ISTRING

finished: BOOLEAN
    -- Whether the client finished

run
        -- Lets the client run
    require
        attached session
    local
        l_fetcher: separate FETCHER
    do
        check attached session as l_session then
            separate l_session as c_session do
                separate c_session.messages as c_messages do
                    create l_fetcher.make (username, c_messages, output)
                    c_session.enter (username)
                    c_messages.extend (username + " has entered the room.")
                    separate l_fetcher as c_fetcher do
                        l_fetcher.start
                    end
                end

                from
                    finished := False
                    last_count := 0
                until
                    finished
                loop
                    read (input)
                end
            end
        end
    end

read (a_input: like input)
        -- Reads from input until there is no more to read
    require
        a_input.count > last_count
    local
        l_count: INTEGER
    do
        from
            l_count := a_input.count
        until
            last_count = l_count or else
            a_input[last_count] ~ "\bye"
```

```
            loop
                if attached session as l_session then
                    separate l_session.messages as c_messages do
                        c_messages.extend (
                            username + " says: " + a_input[last_count]
                        )
                    end
                end
                last_count := last_count + 1
            end
            if a_input.count > last_count then
                c_messages.extend (username + " has left the room.")
                finished := True
            end
        end
end
```

Fetcher

The fetcher fetches the messages from the server to be shown in the client, which includes the messages sent by the client. It stops when it receives the message that the client left the room; no more messages are fetched afterwards. Like the client, the fetcher uses a wait condition based on the last observed count of items in the list.

```
class
    FETCHER

create
    make

feature {NONE} -- Initialization
    make (a_username: ISTRING;
            a_messages, a_output: separate LIST[ISTRING])
        do
            username := a_username
            messages := a_messages
            output := a_output
            last_count := messages.count
        end
feature
    messages, output: separate LIST[ISTRING]
        -- The remote messages an the list that
        -- represents the user-readable output

    username: ISTRING

    last_count: INTEGER
        -- The number of messages on the server
        -- as seen the last time the fetcher checked
```

```
    finished: BOOLEAN
        -- Whether the fetcher finished

    start
            -- Lets the fetcher run
        do
            from
                finished := False
            until
                finished
            loop
                read (messages, output)
            end
        end

    read (a_messages, a_output: like output)
            -- Reads from a_messages and puts it into
            -- a_output a_messages contains no new message
        require
            a_messages.count > last_count
        local
            l_count: INTEGER
        do
            from
                l_count := a_messages.count
            until
                last_count = l_count or else
                a_messages[last_count] ~ (username + " has left the room.")
            loop
                a_output.extend (a_messages[last_count])
                last_count := last_count + 1
            end
            if l_count > last_count then
                a_output.extend (username + " has left the room.")
                finished := True
            end
        end
end
```

Session

Every client receives its own session object. By holding onto a lock on the object until the user logs out, the server is notified if the connection breaks and can run some clean up code.

```
class
    SESSION

feature
    messages: separate LIST[ISTRING]
```

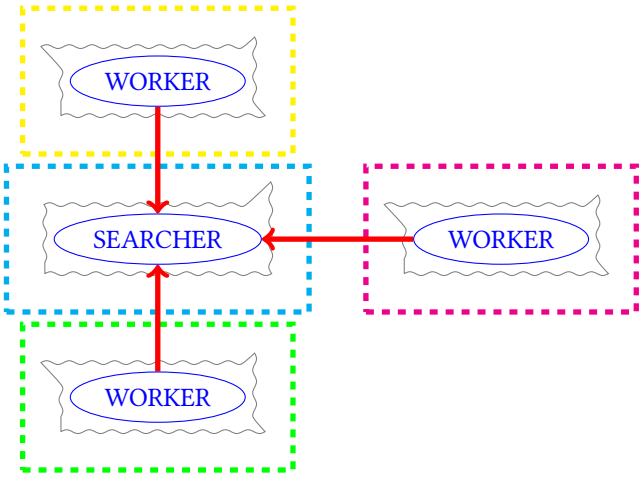


Figure 4.3: Example state of the search engine

```
    username: ISTRING

    enter (a_username: ISTRING)
            -- Sets up the session
        do
            username := a_username
            compensate (agent clean_up (messages))
        end

feature{NONE} -- Compensations
    clean_up (a_messages: like messages)
            -- Notifies others if the client disconnected prematurely
        do
            a_messages.extend ("Connection lost to " + username + ".")
        end
end
```

4.3 Computing Cluster

4.3.1 Architecture

The architecture of the search engine is the opposite of the chat server example, yet similar: A single client, the search engine, has multiple suppliers, the workers. Whenever the search engine has to perform a search, it sends the request to all connected workers, waits for their results and then sorts the results according to their relevance.

This architecture is typical where distribution is used to speed up a calculation by exploiting parallelism or if the memory requirements for the calculation force the usage of multiple machines.

While it depends on the kind of work, compensation is often not required since an incomplete transaction has no adverse effect. In our example, the only effect of a

search is that the `found` variable is set to the result.

4.3.2 Initialization

The search engine connects to all the workers and puts them into a list. The initialization code in this example does not start a server since it is not required for the engine to work. However, a practical implementation would probably start a server to let remote clients make search requests.

4.3.3 Classes

Search Engine

The search engine class acts as the front-end of our search-engine. When it is created, it connects to the various workers that host the data. Its only accessible feature, `search`, distributes the search over all the workers and then returns a sorted list of results. The search engine class can act as a worker too, which allows for a more hierarchical architecture.

```
class
    SEARCH_ENGINE

inherit
    SEARCH_WORKER

create
    make

feature {NONE} -- Initialization
    make (a_worker_addresses: LIST[STRING])
            -- Creates a new search engine that uses the nodes
            -- in a_worker_addresses
        local
            l_dscoop: DSCOOP
            l_index: detachable separate ANY
        do
            create l_dscoop
            create workers.make

            across a_worker_addresses as l_iter loop
                l_dscoop.connect (l_iter.item, 7000)
                l_index := l_dscoop.last_index_object
                if attached {separate WORKER} l_index as l_worker then
                    workers.extend (l_worker)
                end
            end
        end

feature
    search (a_text: ISTRING): LIST[SEARCH_RESULT]
            -- Search for a_text in all connected workers
```

```
        do
            a_workers.start
            Result := recursive_search (a_text: ISTRING)
            sort(Result)
        end

feature {NONE}
    a_workers: ARRAYED_LIST[WORKER]
            -- The databases that contain the data and
            -- perform the actual search

    recursive_search (a_text: ISTRING): LIST[SEARCH_RESULT]
            -- Searches for a_text in a_workers
        do
            if not a_workers.after then
                l_worker := a_workers.item
                separate a_workers.item as c_worker do
                    a_workers.forth
                    c_worker.search (a_text)
                    Result := recursive_search (a_text, a_workers)
                    Result.append (c_worker.found)[3]
                end
            else
                create {ARRAYED_LIST}Result.make
            end
        end

    sort (LIST[SEARCH_RESULT])
        do
            ...
        end
end
```

Worker

The worker either performs the actual search using its indexes, or is a search engine by itself. We only present a deferred class, since the actual search algorithm is outside of the scope of this example.

```
deferred class
    SEARCH_WORKER

feature
    search (a_text: ISTRING): LIST[SEARCH_RESULT]
            -- Search for a_text in the data source
        deferred
        end

end
```

Chapter 5

Distributed Scoop

The first hypothesis of the dissertation is that "a concurrency model based on message passing can be used as a programming model for distributed systems with supplier-aware failure mitigation while maintaining a competitive efficiency".

This chapter demonstrates the hypothesis by describing a system, D-Scoop, that integrates all Scoop guarantees while not relying on shared memory. D-Scoop stands for distributed Scoop and is an application of network objects to the Scoop concurrency model. It uses a messaging protocol internally for all communication between nodes.

We use the term D-Scoop for two things. First, D-Scoop is an extension to the Scoop concurrency model for distributed computing by adding compensation and a message-passing protocol to allow it to work over a network. Second, it is a prototype implementation based on the Scoop support in EiffelStudio. The latter supports the presented semantics with the exception of object export[1] due to limitations in the Eiffel introspection support.

After introducing the necessary technical background of network objects and Scoop (section 5.1), we show how their fusion is realized in D-Scoop, our distributed programming model (section 5.2). We go into more depth on how objects are locked to avoid interference (section 5.3) and how compensation helps in managing failure (section 5.4). We then evaluate our prototype against Java RMI (section 5.9) before discussing some potential topics for future work (section 5.10) and concluding (section 5.11).

This chapter is based on the paper *An Interference-Free Programming Model for Network Objects* [61] by the author of this dissertation.

5.1 From Network Objects to Distributed Scoop

Our work combines networks objects with Scoop. As mentioned in chapter 3, Scoop provides a non-interference guarantee to avoid high-level data races. This section describes an evolution of network objects to Scoop by exploring a simple example.

Our example focuses a transfer feature in an application dealing with bank accounts, similar to the bank account example in chapter 4. It tries to move money from one account to another; if the first account's balance is not sufficient, it displays an

[1] This includes passing user-defined expanded types. However, the D-Scoop library provides an expanded, immutable `ESTRING` class so that strings can be passed by value.

error message. Note that we use the **separate** keyword to make sure the code works with SCOOP, but we do not assume SCOOP semantics at first.

```
transfer (s, t: ACCOUNT; am: NATURAL)
do
    if s.balance >= am then
        s.withdraw (am)
        t.deposit (am)
    else
        -- Show error message
    end
end
```

Listing 5.1: Bank account transfer feature in the client: sequential

Sequential Program. The sequential version of the example is shown in listing 5.1. Assuming that there are no other threads in the system, the feature is correct.

```
transfer (s, t: ACCOUNT; am: NATURAL)
do
    if s.id < t.id then
        s.lock; t.lock
    else
        t.lock; s.lock
    end
    if s.balance >= am then
        ...
    end
    s.unlock; t.unlock
end
```

Listing 5.2: Bank account transfer feature in the client: multi-threaded

Multi-threaded Program. If the accounts can be accessed concurrently, then locks or other measures are required to ensure the atomicity of `transfer`. A possible implementation is shown in listing 5.2: the accounts are doubling as locks and the transfer feature uses them to ensure atomicity while avoiding deadlocks through a global order based on the account ids.

Multi-threaded Program with Network Objects. Multi-threaded programs can use network objects for distribution. This case is similar to one without network objects. However, the implementation of `lock` is more complex as it has to ensure that the lock is released if the client is disconnected before it issues `unlock`. Furthermore, if the client dies before issuing the `deposit` statement, money is lost. There is no simple solution to this without completely restructuring the program.

```
transfer (s, t: ACCOUNT; am: NATURAL)
do
    if !s.transfer_to_if_sufficient (t, am) then
        -- Show error message
    end
end
```

Listing 5.3: Bank account transfer feature in the client: active objects

Active Objects. Active objects do not use locking for synchronization. Instead, every call is handled atomically by the supplier just like a monitor. By restructuring the program, we can adapt transfer to use with active objects. Listing 5.3 shows the transfer feature after the restructuring. We see that instead of withdrawing and depositing, we have a special supplier function to transfer money to a target account. This achieves atomicity and makes sure that the transfer is completed even if the client crashes while the supplier processes the request. However, it also has two drawbacks: It no longer separates queries from commands and it requires adaptation of the supplier. In some cases, the supplier API cannot be adapted, which makes this solution impossible. Note that this version also works in a multi-threaded environment with accounts as monitors.

```
transfer (s, t: separate ACCOUNT; am: NATURAL)
do
    if s.balance >= am then
        s.withdraw (am)
        t.deposit (am)
    else
        -- Show error message
    end
end
```

Listing 5.4: Bank account transfer feature in the client: SCOOP

SCOOP. When implemented with SCOOP, the transfer feature is almost indistinguishable from the sequential version. The SCOOP version in listing 5.4 differs from the sequential version in listing 5.1 only by the usage of the `separate` keyword. The important thing about this is that we can use the same reasoning that we used in the sequential case for the concurrent case: no need to add synchronization primitives as with multi-threading and no need to adapt the supplier as with active objects.

D-SCOOP The goal of D-SCOOP is that it is almost indistinguishable from regular SCOOP. Since regular SCOOP has the goal to allow sequential-like reasoning, D-SCOOP makes it simple to to turn sequential programs into concurrent distributed programs by first going to SCOOP and then D-SCOOP. In fact, the transfer feature is the same in SCOOP as in D-SCOOP. Since distribution always involves failure, the client assumes that the supplier is able to rollback changes if something happens to the client before it finished. Although this does require changes in the supplier, these changes are not client-specific but rather a direct consequence of providing failure tolerance.

5.2 The Distributed Scoop Framework

In this section we present the D-Scoop framework, which combines network objects and the Scoop synchronization semantics into a single, distributed programming model that maintains the simplicity of the original abstractions. We present an overview of its architecture and communication protocol, and explain how separate calls are generalized to potentially remote objects (sections 5.3 and 5.4 describe in more detail how locking of remote objects is achieved in D-Scoop, and how the system compensates for unresponsive clients).

A prototype implementation of the D-Scoop model is available online [18]. Our prototype builds upon the Scoop support for Eiffel in EiffelStudio [20], which implements the model using threads and shared memory. D-Scoop generalizes the implementation, allowing for multiple instances of potentially remote Scoop programs to communicate, under-the-hood, by asynchronous message passing.

5.2.1 Components

D-Scoop consists of several components:

Compiler. D-Scoop requires a modified compiler. The prototype compiler is built upon the official Scoop implementation in EiffelStudio.

Runtime. A D-Scoop protocol needs some additional functionality provided by the runtime environment, which is part of the compiler suite.

Library. Development with D-Scoop requires the interaction between the program an the runtime for initialization and connection management. The D-Scoop library provides classes for this purpose.

Protocol. The D-Scoop nodes communicate with each other using a protocol based on simple message passing. Protocol handling is integrated into the D-Scoop runtime and library.

Language. D-Scoop does not add additional constructs to the language, or modify the language in another way. This is an important aspect of D-Scoop, as it simplifies the step from concurrent to distributed computing. However, we propose a few language extensions for Scoop that are particularly useful in D-Scoop.

5.2.2 Mechanisms

In principle, development in D-Scoop is the same as in plain Scoop as shown in chapter 3, which was the intention, with some added mechanisms for handling failure. When a program starts there is a single region with its processor, the root. More regions/processors can be created, but all of them reside on the same node, just as it is with regular Scoop.

Creation of nodes. An instance of a Scoop program turns into a D-Scoop node when it connects to another D-Scoop node or starts a server that accepts incoming connections. There is no built-in mechanism to create new nodes, instead, D-Scoop operates by connecting existing nodes.

The reason behind this decision is that how nodes are created is highly dependent upon the application, so the framework is not favoring any approach. Creating a new node is basically starting a Scoop program which turns into a D-Scoop node.

Creation of remote objects. The reason why D-Scoop has no primitive for creating remote nodes is a technical one: A remote node runs on another machine. This machine has to be started somehow and a process on this machine has to accept node creation requests. There are many different ways to achieve this and finding one that fits all use cases is nigh impossible. Also, this can be wrapped in a library. This library could start the machine using Wake-on-LAN and wait for it to start the program. In many cases, the machines are started independently, for example a client that is started by a user and then connected to the system.

While the reason for not having a primitive for starting nodes is mainly technical, the reason not to allow remote creation of objects is due to security: it becomes very difficult for devising a security mechanism in the program if other nodes can freely allocate objects. If the creation of remote objects is needed, the factory pattern can be used with the factory object residing on the node where the new object should reside. Furthermore, the usage of a factory allows the node to decide which exact version of a class should be used, a very important prerequisite for upgrading a node in a system without a complete shutdown of all nodes.

By using the D-Scoop library, a program can connect to another node and retrieve a so-called index object. This object resides on the other node and through queries can give access to other objects. This approach differs from RMI and CORBA, which use so-called object registries. If this is needed, the index object can provide the functionality. However, in many cases, simply using an appropriate index object is all that is needed.

Connecting. A connection is established from one node to another. To do this, the address and port of the other node is needed. The simplest way to make a connection is to use the **DSCOOP** class as shown in listing 5.5.

```
local
    l_dscoop: DSCOOP[2]
    l_index: detachable separate ANY
do
    create l_dscoop
    l_dscoop.connect (a_server, 7000)
    l_index := l_dscoop.last_index_object
    ...
end
```

Listing 5.5: Connection to a D-Scoop system

If the connection was not established, the resulting index object is **Void**. Note that this connection is bi-directional, so the other node can make calls to objects it has

access to. However, in order for other nodes to access local objects, a server has to be started, which will be explained shortly.

It is possible that a node does not provide an index object, typically if it is only a service consumer. Explicitly connecting to such a node is usually not necessary since D-Scoop manages connections automatically.

Autoconnect. It is possible that a node receives a reference to an object residing on a node it has no connection to. D-Scoop then automatically establishes the connection to enable communication with this object. This is completely transparent.

Autodisconnect. While it is possible to terminate a connection manually, the usual way is to let D-Scoop handle it, since a premature termination might cause failures. If a connection is no longer in use, that is, there is no reference to an object in any region of the node at the other end and vice versa, D-Scoop terminates the connection automatically.

Servers. To let other nodes connect to the local node, either because the local node acts as a service provider or because it is a peer, the D-Scoop server is needed, as it opens a port for incoming connections. Starting the server using the **DSCOOP** class is shown in listing 5.6.

```
feature -- Server
    run_server
        -- Start the chat server
        local
            l_dscoop: DSCOOP
        do
            create l_dscoop
            l_dscoop.start_server (agent index_object_factory, 7000)
        end

    index_object_factory: ANY
        do
            ...
        end
```

Listing 5.6: Starting a server in D-Scoop

The first argument is the index object factory feature which creates the objects that should given, by reference, to connected processors inquiring about the index object. If given a regular object, references to this exact object will be given upon request for the index object. **Void** can also be given, in which case the clients cannot retrieve the index object. Passing **Void** is useful if a node wants to allow other nodes to autoconnect, but not to offer services itself. This can occur in a client-server architecture where the clients can receive, from the server, references to objects on other clients so that they can communicate directly.

Note that the term *server* as used here can be equated with the more technical term of listening socket. As such, it does not only apply for a typical client-server architecture, although it is impossible to implement a server in this sense without it

setting up a D-Scoop server. While a client, in the client-server sense, not necessarily needs to set up a D-Scoop server, a peer as in peer-to-peer architecture[3] needs one so that other peers can connect.

Architecture. In D-Scoop, nodes are instances of Scoop programs[4]. A D-Scoop system max contain nodes that stem from different programs that share some classes, that is, the classes used remotely. It is not necessary that a client has effective versions of the supplier classes it wants to reference, a deferred suffices. A node can open a connection to another node through a network socket, which is then shared by all of its processors. A node can request the *index object* of another node, which is a regular object that acts as the entrace to the whole API of the node. For nodes implementing simple services, this object may have all the features the node provides. It is also possible let the index object implement some form of object registry that refers to local and possibly remote objects providing services. There are no restrictions to what features an index object exports, which makes it possible for the developer of a system to use a class that exactly fits his or her requirements for the node.

It is valid for a node to not supply an index object, typically if it is a client in a pure client-server style setup. To be able to accept incoming connections from other nodes, a node must start a *server* (see above). To provide an API for clients, the node typically has to provide its own index object, or a factory that generates them. Every node in a D-Scoop network has a unique *identifier* (ID), which is independent of any other IDs such as IP addresses. Object references in D-Scoop include this node ID, along with their object and processor identifiers (as in classical Scoop), with the latter important for determining the number of processors involved in a separate block.

Garbage Collection. Within nodes, we rely on existing mechanisms of Scoop for garbage collecting local objects and processors. D-Scoop however must also account for objects used by multiple nodes. To achieve this, we use a distributed garbage collection algorithm similar to that of Birrell et al. [5].

5.2.3 D-Scoop Protocol.

The nodes in D-Scoop networks communicate, via their connections, using an asynchronous message-passing scheme. Messages conform to a protocol and can be one of two types: a *request*[5] or a *reply*. Requests are sent from a *client* node to a *supplier*, defining work for the supplier to do. Replies are sent back from the supplier to the client to indicate the outcome of a request. Note that some type of requests do not cause a reply, for example the [Pass] request presented below.

Messages in the D-Scoop communication protocol have *subjects* which convey their intended semantics. Messages that are requests can have one of many different subjects which we outline in the following. Replies however only indicate success ([OK]) or failure ([FAIL]), sometimes with additional arguments, such as the result of a query call.

The simplest request subjects are [Hello], [Ping] and [Index], which respectively initialize a connection between nodes, test whether an existing one is still alive, and request

[3] See the distributed banking example in chapter 4 as an example of a peer-to-peer architecture. Also, as architectures can be combined, a client can also be a peer to other clients of the same server if they communicate directly with each other.

[4] In EiffelStudio, these are called processes

[5] Note that these are distinct from the requests used for inter-processor communication in Scoop.

the index object of the supplier node (which typically provides an API of features for retrieving more objects).

A number of requests are required to realize a separate block involving remote objects. A [Prelock] request announces that a processor in a client node wishes to log calls on one or more processors in a supplier node. When a supplier is ready, the client can issue a [Lock] request to announce it is now entering the separate block, upon which the supplier sends back a handle that can be used to log calls. Following this, it can issue requests corresponding to asynchronous feature calls ([ACall]) and synchronous calls ([SCall]), which includes queries. In case of the latter, if a lock needs to be passed, [Pass] messages need to be sent, before the call itself, to the locked suppliers. Both types of call requests cause replies, in the case of asynchronous calls immediately, for synchronous calls when they are completed. To announce leaving the separate block, the client sends an [Unlock] request. (We describe in more detail how these requests acquire locks in section 5.3.)

Requests with the subjects [Share] and [Release] are respectively used for obtaining and revoking permission for given object references to be shared with third party nodes. They are also used by D-Scoop for garbage collecting.

Finally, [Await] and [Ready] requests are used to implement condition synchronization on remote objects. In short: if the condition does not hold, the client processor issues an [Await] request before going to sleep. This instructs the supplier to wake it up with a [Ready] request once the state of the remote objects changes, so that the condition can be checked again.

Detailed description of the messages and their arguments are provided together with the detailed semantics in chapter 6.

Message Handling. Incoming messages are handled by the request handlers of D-Scoop nodes in multiple stages, depending on their subjects. If an incoming message has the subject [Hello], [Ping], [Share], or [Release], then it is handled directly. If a message is a reply, then it is relayed to the appropriate processor within the node.

For messages concerning separate blocks and condition synchronization, a more careful treatment is required. In D-Scoop, every node has specially designated *proxy processor* per active connected processor for handling incoming lock and call requests, as well as to provide a region for *proxy objects*, which are surrogates (or placeholders) for actual remote objects, holding references to them. Proxy processors are only needed for remote processors that either currently have or seek a lock on a local processor, or for remote regions that contain an object a local processor has a reference to. This additional layer is used to catch special contexts in which calls are treated differently, including separate callbacks (see [49]), and a Scoop extension for data objects (see [46]). The latter are regions without a processor, requiring the usage of the client's processor for execution of any requests. Since the client processor is located on another node that does not share memory with the supplier, this is not possible without the use of a proxy processor on the supplier's node.

To minimize the overhead of proxy processors and objects, they are created only when needed and destroyed when they are not. For example, if not existing already, receiving a [Prelock] request with some given processor identifiers will trigger the creation of a proxy processor on that node. Or when receiving a result from a separate call with a reference to a remote object, a proxy object is created and if necessary, a proxy processor and region to hold it. When proxy objects are no longer in use by local processors, they can be collected by the local Scoop garbage collectors, which

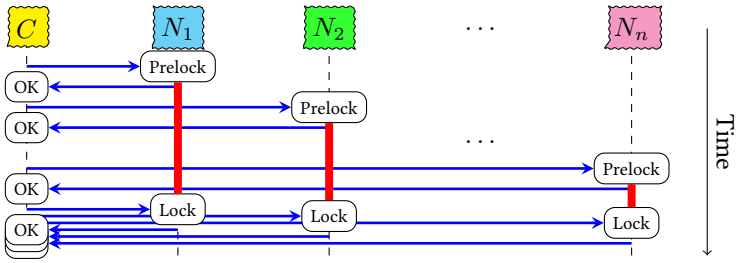


Figure 5.1: Prelock phase: a processor on node C is entering a separate block involving `separate` objects on remote nodes $N_1, \ldots N_n$

in turn may cause the associated proxy region and processor to be removed.

Remote Calls in Separate Blocks. The communication protocol presented is ultimately the glue that allows for network objects to be used within the Scoop framework. Our aim was to make the fusion of these concepts as seamless as possible: programmers should not need to be aware of the communication protocol for network objects, and the core abstractions of Scoop should not need to be fundamentally reinvented to accommodate the extension.

In D-Scoop we were able to maintain the original abstractions provided by separate blocks, while also providing a natural generalization to support objects residing on other nodes. When a processor needs to make a call on a `separate` object, there are now three cases to distinguish. If the target object shares the same processor (and thus the same node), the call is executed immediately—as in Scoop. If the target object has a distinct processor but on the same node, the processor logs a request in a private subqueue for the caller (see Section 5.1)—as in Scoop. If the target object has a distinct processor on a remote node, however, the D-Scoop communication protocol comes into play, and a ACall or SCall message is sent to to the remote node.

5.3 Locking Remote Objects

We have presented an overview of the D-Scoop architecture, its messaging protocol, and its generalization of `separate` blocks to support calls on remote objects. In this section, we describe how locking of remote objects and thus distributed separate blocks are achieved.

In D-Scoop, separate blocks are handled in three phases: (i) the *prelock phase*, for ensuring a correct ordering; (ii) the *issuing phase*, for enqueuing calls; and (iii) the *execution phase*, for executing calls. The issuing phase happens strictly after the prelock phase. While the execution phase cannot start before the issuing phase, the two can otherwise overlap due to asynchronicity.

Prelock Phase. In standard Scoop, if a processor enters a separate block, the processors handling the `separate` objects generate private subqueues for logging calls (see Section 5.1 and Figure 3.1). In D-Scoop however, if a processor enters a separate block involving `separate` objects on remote nodes, messages must be sent to trigger the generation of subqueues in a way that preserves the usual reasoning guarantees. We refer to this messaging phase as the *prelock phase*.

A client node seeking to enter a separate block involving remote objects must first announce its intention by sending [Prelock] requests to the nodes they reside on. This is done in a fixed order (a global order based on node IDs) to avoid deadlocks, and one-at-a-time; an [OK] reply must be received before the next [Prelock] is sent. Once the last such request is successful, the client node announces that it is entering the separate block and will start issuing calls. This announcement is made via [Lock] requests, which can be sent asynchronously in any order. By replying with [OK], the supplier nodes are acknowledging that the involved processors have created private subqueues and are ready to enqueue calls from the client. Figure 5.1 exemplifies this phase for a client node C that wishes to enter a separate block involving remote objects on supplier nodes $N_1, \ldots N_n$. Here, an arrow denotes the transmission of a message, with its subject given at the end (additional parameters are not visualized).

When multiple nodes are entering prelock phases involving common supplier nodes, blocking must occur in order to maintain the separate block order guarantees. In particular, if a [Prelock] message is sent but the supplier is already involved in the prelock phase of a competing node, then the system blocks on that message. Instead of blocking for the whole of the competing node's separate block, D-SCOOP permits a more fine-grained and efficient solution. In particular, it only blocks until the competing node leaves its prelock phase and starts issuing calls. That is to say, D-SCOOP only blocks while "setting up" the subqueues in a correct order; competing issuing phases can otherwise safely run concurrently.

Note that the seamless integration of D-SCOOP in SCOOP entails that local clients are treated the same as remote clients to avoid deadlocks and starvation, even if they do not send messages over the network. Also, the prelock protocol ensures that all private queues are put into the queue-of-queues atomically. Earlier publications might imply that the insertion of the private queues are done independently, which would introduce deadlocks in some cases and the possibility for a client to observe one supplier as it was before the execution of a separate block from another client and a second supplier as after the execution of the same block. The current SCOOP implementation in EiffelStudio already ensures the atomicity of private queue insertion, and D-SCOOP keeps this guarantee.

Issuing and Execution Phases. The prelock phase ends and the issuing phase begins when the final [Lock] request is successful. At this point, the processors handling all the involved remote objects are ready to enqueue calls. In most circumstances, commands on remote objects are requested via asynchronous [ACall] messages, and queries are requested via synchronous [SCall] messages. The supplier nodes enqueue commands and immediately reply with an [OK], but the client is not required to wait for the [OK] before continuing. When a query is received, however, the supplier node enqueues it, but only replies once it has been executed (passing the result in an additional parameter of the [OK] message).

The execution phase begins with the execution of the first logged call. If all the calls are asynchronous, it can take place strictly after the issuing phase. The issuing phase ends on sending the [Unlock] message; the execution phase ends on processing it.

Example Communication. We return to our running bank account example, to which we add a simple feature `withdraw_from` (listing 5.7) for withdrawing a given amount from a given account that we assume to be remote. The feature first synchronously queries the remote object to check that the balance is sufficient, before

```
withdraw_from (s: separate ACCOUNT; am: NATURAL)
do
    if s.balance >= am then
        s.withdraw (am)
    else -- Notify user
    end
end
```

Listing 5.7: Client feature to withdraw money from a account

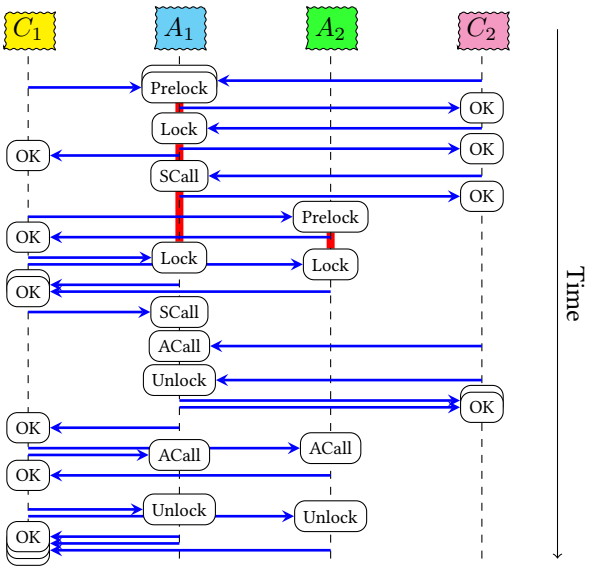


Figure 5.2: All three phases: a processor on C_1 calls `transfer` on A_1 and A_2; a processor on C_2 concurrently calls `withdraw` on A_1

asynchronously decreasing the balance.

Suppose we have a running D-Scoop system with two bank accounts on different nodes (A_1, A_2). Suppose now that a client node (C_1) is trying to `transfer` an amount from A_1 to A_2, while another client node (C_2) is trying to `withdraw` an amount from A_1. Recall that the bodies of both features are separate blocks (involving, respectively, **`separate`** accounts on A_1, A_2 and A_1). Figure 5.2 visualizes the messages exchanged in one possible behavior.

Observe that both clients initially send a [Prelock] request to A_1. The request from C_2 is received first and is therefore answered immediately; meanwhile, C_1 blocks. Since C_2 only tries to lock a processor on A_1, it proceeds to send a [Lock] request, thus completing its prelock phase and generating its private subqueue on A_1. This allows C_1 to unblock and its first [Prelock] request finally succeeds.

Since the prelock phase of one client can take place in parallel to the issuing and execution phases of another, C_2 already starts issuing calls before C_1 concludes its prelock phase. In particular, it requests the `balance` query (via [SCall]) which is executed synchronously (and the balance amount returned). Following this, C_1 requests a [Prelock] on A_2 (which is uncontended), before completing its prelock phase by sending [Lock] requests to A_1 and A_2.

At this point, C_1 issues a `balance` query as it is evaluating its conditional guard. At

the same time, C_2 requests an asynchronous command (ACall) to update the balance, and, upon receiving the OK, exits its separate block via an Unlock request. Since C_2 completed its prelock first, its private subqueue on A_1 is ahead of the subqueue for C_1, and so its call is executed first. Once acknowledged, C_2 knows that the whole transaction (`balance` and then `set_balance`) was successful, and its effects become visible to other clients. Once the OK corresponding to its earlier SCall arrives, C_1 can resume issuing the remaining calls in its separate block before exiting via Unlock requests to A_1 and A_2.

Note that the reasoning guarantees of the separate blocks have been maintained. The calls are executed in program text order and without intervening calls from other nodes: within a separate block, multiple `balance` calls in sequence thus always return the same result. The combination of the prelock phase and the underlying queue of queues semantics prevents the possibility of interleavings that break this.

5.3.1 Lock Passing

A powerful mechanism in SCOOP is lock passing, explained in [49]: clients let the supplier use their locks on synchronous calls to avoid deadlocks. Otherwise, if the completion of the synchronous call includes a synchronous call to one of the processors locked by the client, a deadlock occurs. The SCall message includes a list of locks that the supplier can use, including one on the client for issuing separate callbacks. This is possible because a lock in SCOOP is merely a right to enqueue calls. However, to do this, a client has to send Pass messages to all other remote suppliers, notifying them of the change of ownership of the lock. Conversely, the supplier receiving the locks has to send Unlock messages to pass the lock back. Locks can be passed multiple times. An example for lock passing is given later in section 5.7.

D-SCOOP also supports callbacks: On a separate synchronous call, the client creates a new subqueue at the front of its queue of queues and sends a lock on this queue along with the call. The supplier can now use this subqueue for callbacks. Before returning the synchronous call, the supplier also unlocks this queue.

5.4 Compensating for Failure

Our presentation of D-SCOOP has thus far focused on the challenge and intricacies of combining the network objects abstraction with a concurrency model and runtime. In this section, we turn our attention to a topic that cannot be ignored in the setting of distributed computing: coping with failure.

While failure can often be managed simply, failure in the middle of a separate block, when only some of the commands with side-effects have been issued, needs a more elaborate solution. We introduce *compensation*, D-SCOOP's mechanism for reacting to such failure, and demonstrate its use on our running example.

Compensation. In D-SCOOP, upon failure of a supplier, the client is informed using exceptions, and can react to it appropriately in a **`rescue`**-clause, as per normal Eiffel exception handling, which is similar to exception handling in other object-oriented languages. However, the suppliers in separate blocks are in general oblivious to the status of the client. Our solution is to introduce *compensation*, a supplier-side mechanism for reacting to client nodes that become unresponsive or disconnect prematurely.

The technique registers user-provided closures on suppliers that, before releasing objects locked by disconnected clients, are executed to restore consistency.

The basic technique is adapted from the well-established usage in transactions, in particular, for recovering from long-running transactions or transactions with side effects [9]. It fits naturally with the D-Scoop model, given that features and separate blocks are transaction-like in the sense that other clients cannot observe **separate** objects in intermediate states. One can think of a Lock and Unlock pair as marking the beginning and end of a transaction; after Unlock is processed, all changes become visible.

The scope of compensation is the issuing phase, and encompasses all executed calls on processors that have been acquired during the prelock phase (and only those processors). In the case of nested separate blocks, the outer block has to take into account that the effects of the inner block on arguments not locked by the outer block are already visible if an Unlock was issued. This is different to most definitions of nested transactions, in which the inner transaction always finishes together with the outer transaction. This behavior is required to affect the outside world in long-running transactions, for example to give updates on the status. The traditional behavior would lead to more complexity and break with the Scoop model, which we want to avoid.

Defining Compensation. Compensation closures are provided by the user by calling `compensate` with the closure as an argument[6]. It is possible to define them in the client or the supplier. A client-defined compensation closure is registered before the call to the feature to be compensated (and is ignored by the supplier if no request follows). A supplier-defined compensation closure is provided within the called feature. The latter comes with the advantage that compensation is defined together with the feature, but the former allows for more flexibility: different compensations can be defined depending on where the call is made, which is particularly useful for features that do not always need compensation.

Consider the simple feature `withdraw` for bank account objects (Listing 5.8) which deducts an amount from the `balance` of an account. The listing also includes examples of how to make it compensable. On the left is a snippet of the body of `transfer`, now annotated with client-defined compensation before the call[7]. On the right is supplier-defined compensation, provided at the beginning of the feature body. In both cases, the compensation will restore the old balance if called.

Implementing Compensation. Upon receiving a Lock request, a supplier node stores the IDs of the newly requested processors in a stack. This stack is mainly used to identify which processors need to be released upon Unlock. Each of the processor entries also contains a reference to a list of compensation closures. This list is populated with the closures (agents) registered for compensation. Whenever a processor is unlocked normally (i.e. not due to premature disconnection) the respective list is cleared. However, if a client node disconnects prematurely, the closures in all lists associated with the client are executed in reverse order.

Figure 5.3 shows the call stack caused by a remote client calling the feature `a` and then `h`. The targets of `a`, `b`, `c`, `d` and `h` are owned by processor P_1, while the targets of the calls `e`, `f`, and `g` are owned by processor P_2. During the execution of `c`, P_1 locks

[6] We remark that closures are given with the Eiffel keyword **agent**, and can refer to existing features

[7] The current prototype does not yet support client-defined compensation, see section 5.10.

(a) Client-defined compensation

```
transfer (s, t: separate ACCOUNT;
    am: NATURAL)
do
    if s.balance >= am then
        s.compensate (agent
                s.deposit (am))
        s.withdraw (am)
        t.compensate (agent
                t.withdraw (am))
        t.deposit (am)
    else -- Notify user
    end
end
```

(b) Supplier-defined compensation

```
withdraw (am: NATURAL)
    require
        am > 0
    do
        compensate (agent
            deposit (am))
        balance := balance - am
    end

deposit (am: NATURAL)
    require
        am > 0
    do
        compensate (agent
            withdraw (am))
        balance := balance - am
    end
```

Listing 5.8: Adding compensation to the transfer example

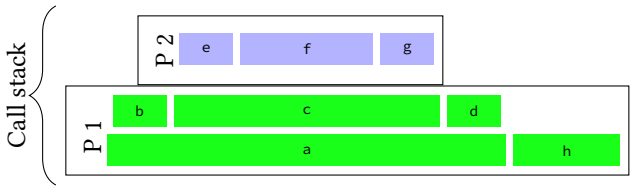


Figure 5.3: Example call stack

P_2 to execute. After a is finished, the client sends another request to execute h before releasing P_1.

We now take a look at three failure scenarios, all of them due to a premature disconnect by the client. If the client disconnects before a is executed, nothing happens. The client's lock on P_1 is simply lifted. The second case is more complex: if the client disconnects while a is executing, the calls a, b, ... g are all executed as requested. Since P_1 is issuing the Unlock request to P_2 before finishing itself, the changes done by e, f, g are visible. The disconnect then causes the compensation closures of d, c, b, a to be executed before the lock on P_1 is released. Consequently, the compensation of c has to deal with the fact that the changes due to e, f, g are already visible.

If the client issued the call to h but got lost before sending the Unlock request, the situation is similar, with the one difference being that the compensation of h is executed before the others.

Compensation and Exceptions. The avid reader may ask whether there is a connection between compensations and exceptions, and indeed there is. Not only are exceptions and compensation two sides of the same coin, exception handling can be seen as a special case of compensation: the **rescue** clause can be seen as a closure, registered at the start of a routine, with the only difference that the closed over state is

the state when an exception is received, not the state when the closure was registered. With this in mind, it would be possible to replace or supplement exception handling with a compensation mechanism: compensations are registered as normal, but in addition to compensations, checkpoints are added, for example by using the `restart` instruction. If an exception occurs, all previously registered compensations are run until the last checkpoint. Normal execution then continues with the instruction directly following the `restart` instruction. Nevertheless, on a premature disconnect, all compensations need to be run that are registered due to the disconnected client.

5.5 Wait Conditions

A client[8] can issue an [Await] message instead of [Unlock] to unlock a processor. In this case, the supplier will send a [Ready] message to the client when its state changed. This allows the client to re-evaluate the wait condition only if there is a chance it may succeed.

The simplest implementation and the one used in our prototype sends the [Ready] messages whenever a client sent [Unlock]. This assumes that wait conditions do not change state or at least not in a way that triggers other wait conditions.

5.6 Passive Regions

Passive regions [46][9] have no processor of their own. If a client issues a call to an object residing in a passive region of another node, the proxy processor applies the call, which causes asynchronous calls to not be synchronized as if the target was on the same node. We chose to do this because it is difficult for a remote client to know whether a region is passive, as this is something that is determined at region creation. Clients may not create remote objects, and therefore also no remote regions, so the information whether a region is passive lies solely with the node that contains the region. With this behavior, a client can assume that all remote regions are active.

5.7 Examples

To give a more complete illustration of the D-Scoop protocol, we present a possible exchange of messages based on the examples in chapter 4.

5.7.1 Chat System: Message Exchange

We assume that two chat partners, yellow and green, connect to a server with color cyan. After a few messages, the yellow client loses its connection. Upon seeing that, the green client also leaves. Due to space constraints, we omitted the input and output processors and their communication.

[8] In this case, the client is the processor of the region that holds the object with the feature that has the wait condition which failed.

[9] Not to be confused with term passive objects as used by some active object models — in Scoop for Eiffel, expanded objects are the closest thing to a such a passive object, that is, an object passed by value to other processors.

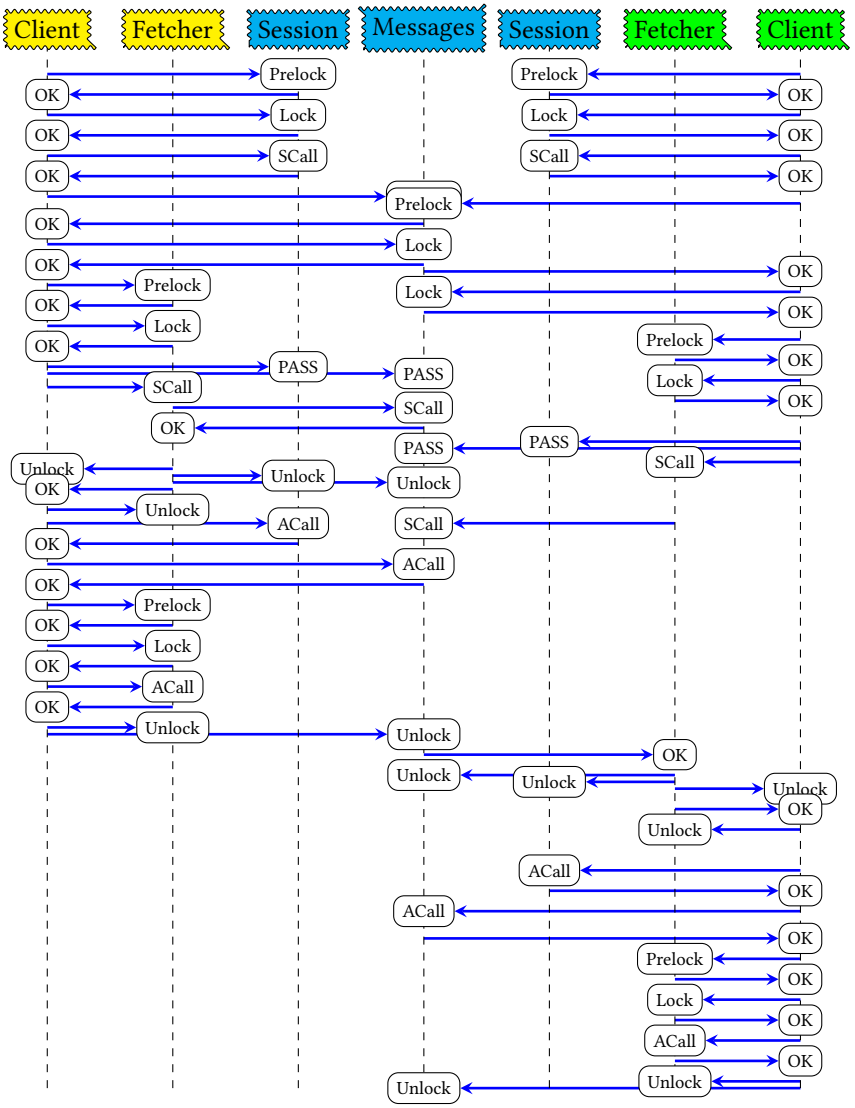


Figure 5.4: Chat server: Exchanged messages – Entering

Entering. The exchanged messages of the first part of the example are shown in fig. 5.4. Both clients connect to the server simultaneously. They receive their own session objects, which have independent processors. Both clients acquire a lock on the session and query it for a reference to the messages list, which they immediately try to lock. The yellow client gets its Prelock message in first, so it also is first to receive the OK. The following Lock request from the yellow client allows the green client to continue, while the yellow client acquires a lock on the fetcher processor to create the new fetcher object. The creation call is synchronous because it passes the locks on both the session and the messages processors. The creation call uses the passed lock on the messages processors to make a synchronous call for the current `count` before sending the Unlock requests to all its locks. Meanwhile, the green client

does the same, but the green fetcher's synchronous call request is stalled because the messages processor is still processing the frame created by the Lock request of the yellow client: the Unlock request from the fetcher was only undoing the lock passing, not closing the frame.

Before this happens, the yellow client still needs to do some things. The first is to release the lock on its fetcher. It then calls `enter` on the session, which is asynchronous. Another asynchronous request to `messages` adds the announcement. Finally, the client acquires a lock on the fetcher again to call `start`, before unlocking the fetcher and the messages.

Now the green fetcher and client can continue. They take the same steps as the yellow client and fetcher before.

Chatting. Figure 5.5 presents the message exchange for a simple chatting sequence. It can be seen as a direct continuation from fig. 5.4.

Both fetchers try to gain a lock on the messages list. The yellow is slightly faster with sending its Prelock message and goes first. Both then issue the synchronous calls for checking the precondition, since yellow was faster during prelock, it receives an answer immediately. It fetches the `count` again and then two messages from the list before unlocking, upon which the green fetcher continues and does the same. Immediately after unlocking, the yellow fetcher requests a lock again, with which it checks the precondition. Since there has been no change, the yellow fetcher issues an await, which unlocks the messages.

Now the clients come into play. They both lock the messages, but this time green goes first. It adds a single message and unlocks. However, the yellow client can do the same at the same time since adding a message is an asynchronous call. While the asynchronous call is executed, the message list sends the Ready message to the yellow fetcher, which causes it to lock the messages, and then fetch the new arrivals.

Disconnect. The last part of the execution is disconnecting, which is shown in fig. 5.6. Both fetchers check the messages list for new messages. The yellow fetcher goes first, but disappears together with the yellow client, which is detected by the messages processor. It reacts by letting the green fetcher continue. The session processor for the yellow client also detects the failure, upon which it locks the messages to add a notification about the disappearance. However, it takes the green fetcher a second round to also get this message, since the actual execution of the asynchronous call that adds the message is deferred until the green fetcher unlocks. The green client now makes a clean exit, which the green fetcher notices.

5.7.2 Distributed Banking: Message Exchange

We assume a system like in fig. 4.1 with three branches:

1. The yellow branch located in the upper left corner,
2. the green branch located in the upper right corner, and
3. the lower magenta branch.

Our focus is a transfer operation by one of the magenta agents from a yellow to the green account. Figure 5.7 shows a possible message exchange between the different processors.

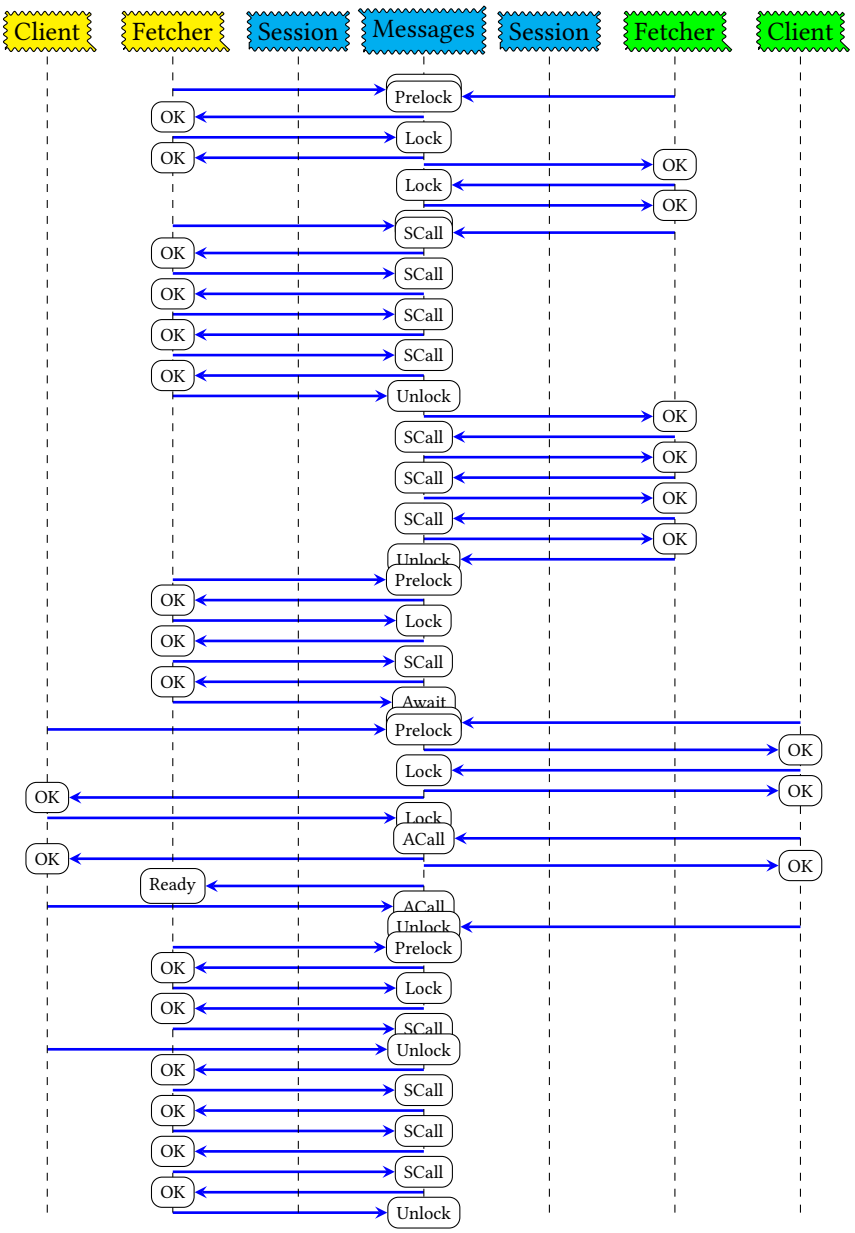


Figure 5.5: Chat server: Exchanged messages – Chatting

The execution of `transfer` starts by calling `find_account`. We assume that the lock on the register was not acquired already, so the agent sends first a prelock and then a lock message to the processor of the register.

The first query to the register is to look up the first account. Since the account is located on another branch, yellow, the register of the magenta branch has to send a query to the yellow register. To do this, it first needs to acquire a lock, which is achieved through the prelock protocol. When the magenta register receives the result, it unlocks the yellow register and returns it to the agent, which in turn unlocks the

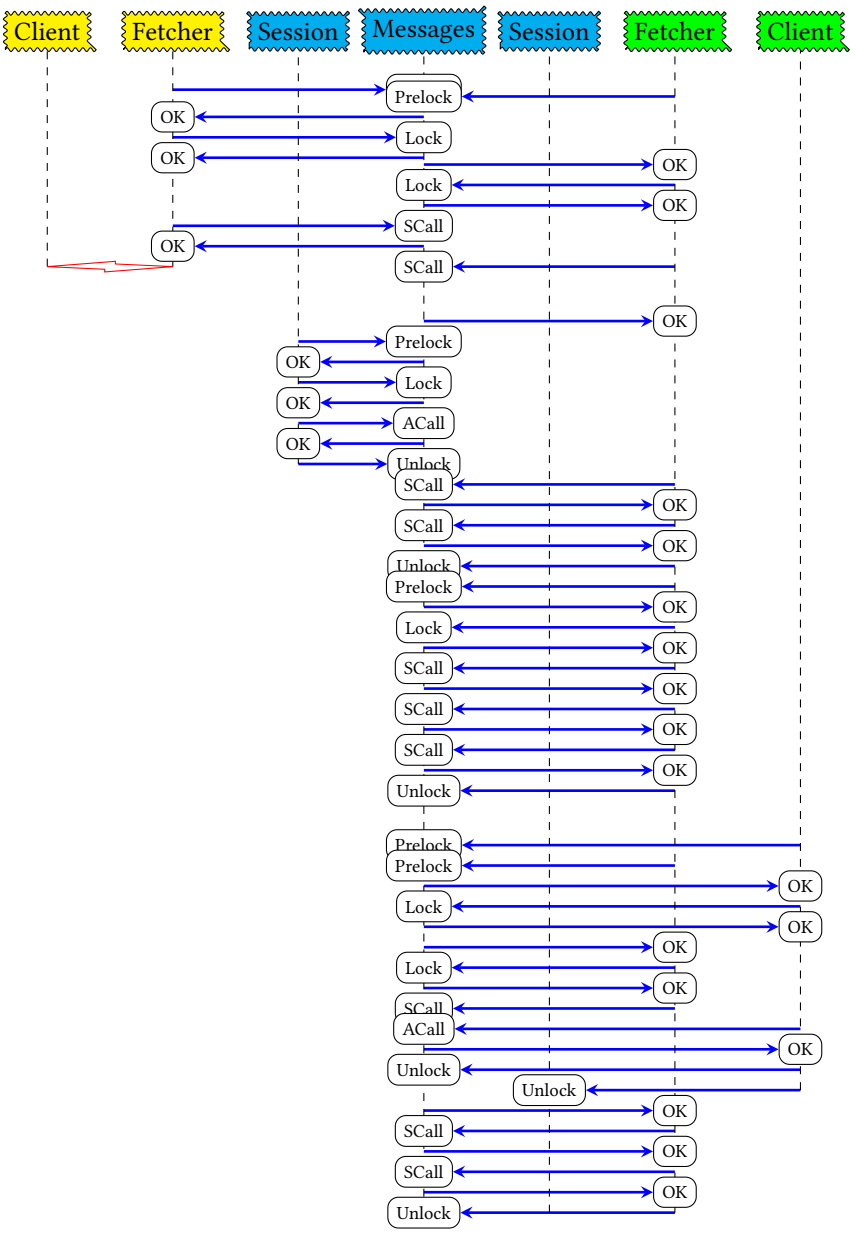


Figure 5.6: Chat server: Exchanged messages – Disconnect

magenta register. The same procedure is done for the yellow account.

With references to both accounts, the agent acquires locks on their processors. It then sends a query to the yellow account for checking its balance. Assuming that the balance is sufficient, two asynchronous calls, one for deposit, one for withdrawal, are sent before unlocking the processors of the accounts.

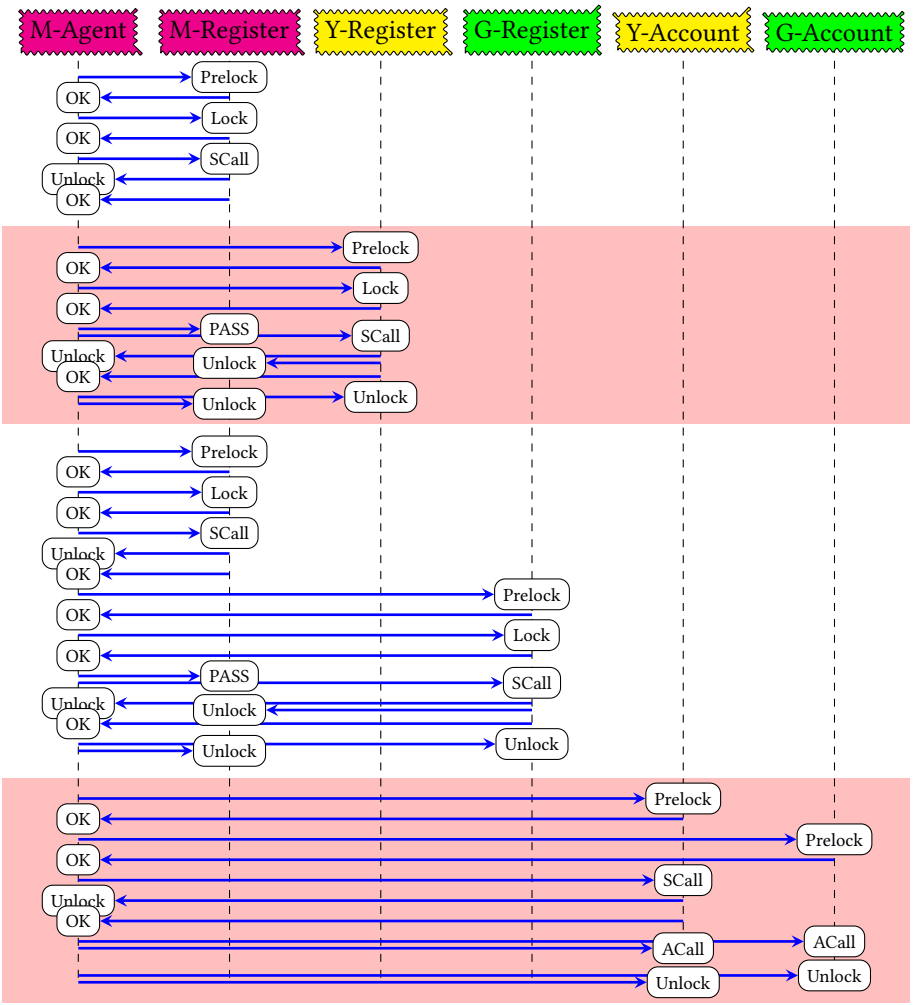


Figure 5.7: Distributed Banking: Exchanged messages

5.8 Semantics

Due to their complexity, the formal semantics for the D-Scoop system, especially the protocol, are presented separately in chapter 6.

5.9 Evaluation

We evaluated D-Scoop against Java RMI to gauge its performance against a well-established and widely used approach based on network objects. We sought to collect evidence towards answering two questions. First, is there a performance overhead associated with the automatic synchronization in D-Scoop, and does it become incommensurate with the effort to manually write synchronization code? Second, do the language abstractions of D-Scoop facilitate simpler code?

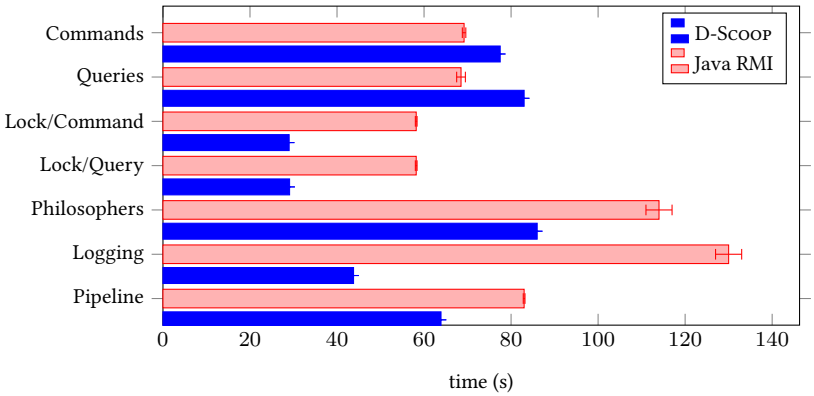


Figure 5.8: Benchmark results: each run involved several thousand iterations

Example Selection. D-Scoop and Java RMI differ in many aspects: not only in the model, but also in terms of the underlying programming languages (Eiffel and Java) which have many points of variation regarding performance and compilers. In this context, we devised a set of four microbenchmarks geared towards comparing the performance of calls: (i) *command call*, in which a single client sends a series of command calls to the supplier; (ii) *query call*, analogous, but with query calls; (iii) *locking and command call*, in which a few clients compete to lock a supplier object and send a single command call; and (iv) *locking and query call*, analogous, but with a single query call.

In addition to microbenchmarks, we also evaluated D-Scoop against Java RMI on three larger examples. First, *dining philosophers*, a classical example where multiple objects (forks) are repeatedly locked. For this benchmark, all philosophers and forks reside on different nodes, and we assume that eating, using the fork, and thinking take no time. A second, more practical example: a *log server*, in which various events are logged. Here, there are multiple log servers for redundancy, meaning that copies of logs can still be retrieved if one fails. To ensure a consistent ordering across servers, a client must lock all of them before adding the entries. In our benchmark, three clients repeatedly generate a simple log message, gain locks across the servers, and then place it. Third, a *pipeline* representing distributed services. Each stage waits until the previous stages are ready before retrieving data and processing it. Each stage provides one operation of the well known formula $\sqrt{a^2 + b^2}$. We measured the time the final stage needed for a specific number of calculations.

For Java RMI, explicit locking was used to establish a comparable flexibility in the clients. Furthermore, the Java code explicitly orders the locks so as to avoid deadlocks. The source code of the examples and of D-Scoop itself can be found on our supplementary material webpage [18]. We did not compare the performance of local suppliers, since such a comparison would be skewed: local suppliers in D-Scoop automatically use shared memory for communication. In Java RMI, the communication still goes through the network stack, which adds significant delay. If, in the other hand, Java RMI would be entirely skipped and the supplier object used directly, Java RMI would be significantly faster, but this is no longer a distributed system.

Performance. Overall, we found that despite the potential overhead of automatic synchronization, D-Scoop's performance is competitive with—and can be superior

Table 5.1: Code complexity

	Classes		Features		Instructions	
	RMI	D-Scoop	RMI	D-Scoop	RMI	D-Scoop
Microbenchmarks	3	2	8	6	19	13
Dining philosophers	3	2	6	3	18	10
Logging	6	3	16	9	23	10
Pipelines	2	1	10	16	62	42

to—explicit locking-based synchronization in Java RMI. The results of the performance evaluation are listed in fig. 5.8 and are the averages of 30 runs; we used two off-the-shelf laptops connected by an ethernet cable. The microbenchmarks show that the performance of both D-Scoop and Java RMI is similar when just issuing commands or queries. D-Scoop commands are a bit quicker than D-Scoop queries due to them being asynchronous, whereas in RMI both are synchronous. When it comes to the locking microbenchmarks, the built-in synchronization in D-Scoop allows for a more significant improvement in speed, both for synchronous and asynchronous calls. However, the synchronization overhead prevents the asynchronous advantage of Lock/Command translating into faster performance than Lock/Query.

For both the dining philosophers and the logging example, the fact that the prelock phase can be done in parallel with the issuing and execution phase of another client proves to be a significant advantage in comparison to RMI[10]. In addition, the logging example shows the advantage of asynchronous calls in D-Scoop. The underlying semantics make it possible to ensure locking of multiple nodes and have multiple clients issuing asynchronous calls at the same time. The pipeline example has less congestion around the synchronized objects; here, the advantage of D-Scoop lies solely in slightly fewer messages sent due to more powerful synchronization mechanisms.

Simplicity. Our second question asked whether the language abstractions also yield simpler code. For our seven benchmarks, we recorded: (i) the number of classes involved, excluding primitive types, classes, and strings, and ignoring the RMI remote interface; (ii) the number of features (i.e. attributes and routines), ignoring the Java "getters" in RMI since they just return an otherwise counted attribute; and (iii) the number of written instructions, excluding boilerplate code. This ensures that the differences are only due to synchronization. Table 5.1 lists the results.

As can be seen, the solutions in D-Scoop are much more compact across the three measurements. In the case of advanced techniques such as condition synchronization — an in-depth discussion is omitted for brevity — the complexity of RMI increases further still. Note that not included in the RMI examples are compensation and the automatic releasing of locks, since they are difficult to achieve in that framework. Also, although the usage of a lock or semaphore is counted as a class, its features are not counted in the feature column since they are already provided by the library. We remark that these numbers only indicate that D-Scoop programs are more compact than their RMI counterparts. What we leave to future work is a study of users themselves to determine whether the D-Scoop abstractions are easier to read and program with, regardless of their compactness. (An existing Scoop study is encouraging [48].)

[10]This is similar to the effect of using Queues-of-Queues in comparison to the old Scoop semantics.

5.10 Future Work

There are still areas for improvement in D-Scoop and Scoop in general.

5.10.1 Creation of Remote Agents

At the moment, the `agent` expression does not work with a remote target, since the expression produces a local object that then gets relabeled to the target processor. With a remote target, the agent should be created on the target node instead, or exported after creation. Without the creation of remote agent, the usefulness of client-side compensations is limited.

5.10.2 Export

Object export can be seen as the client equivalent to object import, a Scoop pattern introduced by Schmocker et al. [62]. With object export, the client sends the object by value instead of by reference, the latter being the default for non-expanded objects. The object and all objects directly or indirectly reachable through non-separate references are copied to the receiving processor, even if it is the same processor as the current processor.

Transparent and explicit export. We can distinguish two forms of export: transparent and explicit. Transparent export is used when passing expanded or immutable (chapter 8) objects.

By not following separate references, the export mechanism is not the same as a deep copy. If the object refers to an object through both a separate and a non-separate reference, the non-separate reference points to the copy but the separate reference keeps pointing to the original. Object export is especially important for distribution since it avoids sending unnecessary messages required by object import.

For example, we assume that the `put_integer_array` feature of a class called `MATH_FORMATTER` takes a separate array of integers as an argument:

```
feature
    put_integer_array
            (a_arr: separate ARRAY[INTEGER])
        deferred
        end
```

Now a client wants to give the formatter such an array for printing. The reference to the formatter is `separate`, so simply passing the array would cause callbacks for every item in the array. The client therefore decides to `export` the array:

```
feature
    data: ARRAY[INTEGER]
    print (a_format: separate MATH_FORMATTER)
        do
            a_format.put_integer_array(export data)
        end
```

A similar mechanism named *deep import* was introduced by Nienaltowski for handling non-separate references in expanded types [49]. The advantage of an explicit

export is that the supplier can be implemented in a more generic fashion: with deep import, it has to accept an expanded argument that wraps the array.

Object export also allows the client to call a feature that expects a non-separate argument with a, which is, except for passing `Void` or a reference the compiler can statically infer to reside on the target processor, otherwise impossible. In other words: when the argument is exported, it conforms to both separate and non-separate formal arguments. The export keyword on an expression of expanded type is redundant, as expanded objects are always exported.

5.10.3 Smarter Wait Conditions

A way to reduce the amount of redundant wait condition evaluations is to replace the simple mechanism to evaluate when to send a [Ready] message with one where processing an [Unlock] or [Await] message triggers [Ready] message if the executed code made at least one attribute assignment to an existing object. A further step is then to record the accessed attributes before an [Await] and trigger the [Ready] message if one of these attributes changed. It has to be evaluated whether the recording of accessed and modified attributes outweighs the benefits of fewer retries.

5.10.4 Explicit Synchronization

In section 3.3 we mentioned a list of conditions for asynchronous calls. This behavior of automatically deciding whether a call is synchronous or asynchronous has the drawback that it cannot be decided statically in all cases. A tool that helps in this case was presented by Meier [40], and it is obvious from the complexity that this is a real problem. It is therefore useful to have a mechanism that ensures that a call is asynchronous by never performing lock passing even if the arguments are locked. While the runtime environment of D-Scoop is ready for this, it requires a language change, for which we propose using the exclamation mark instead of the dot to ensure asynchronicity:

```
target!asynchronous_call (...)
target.regular_call (...)
```

The regular syntax using a dot would not be affected. A problem arises if the target is non-separate, in which case the call cannot be asynchronous. We see three possible solutions of this problem:

1. The call is executed synchronously
2. An exception is raised
3. The call is added for execution later

While the first one is simplest, it again some ambiguity. The other two are consequences on the decision whether asynchronous self-calls should be valid. The second is the result of making it invalid, while the third is the result of allowing it. Note that it is possible to perform a self-call by bouncing it back from another processor by using an agent, and this is presented as a pattern in [62]. The third gives the programmer a simple mechanism to make asynchronous self-calls. Asynchronous self-calls can be used for processing the calls from other clients before continuing. As an example, we assume the features in listing 5.10 are part of a producer of a producer-consumer relationship. It is possible to implement `produce` with a loop, this would make the

producer unstoppable: while the loop is executing, a `stop` request cannot be applied to flip `stopped` and the loop executes until the `stopped` flag is not set the loop keeps executing. From a practical point of view, having a lightweight method, that is, one that does not require an additional processor just to bounce back the call, for asynchronous self-calls is useful.

```
feature
    stopped: BOOLEAN

    produce (a_buffer: separate BUFFER)
        require
            stopped_or_can_produce: stopped or else not a_buffer.is_full
        do
            if not stopped then
                a_buffer.put (create {ITEM})
                Current!produce (a_buffer)
            end
        end

    stop
        do
            stopped := True
        end
```

Listing 5.10: Example: asynchronous self-call in a producer

5.10.5 Asynchronous Replies

It is possible to integrate asynchronous replies in D-Scoop by leveraging agents. This enables completely asynchronous behavior which is useful in some settings, especially if the latencies are high. For example, the following code

```
supplier.run_query (argument) then agent recipient.a_command (?)
```

would request the handler of `supplier` to apply the query `run_query` and then invoke the agent `recipient.a_command` with the result as an argument. This could even be chained using multiple `then` clauses. The asynchronous reply construct implies that the original request is always asynchronous, that is, no lock passing takes place. Combined with the suggestion above, this means that the dot on the left hand side should always be replaced by an exclamation mark for consistency.

The call on the right hand side could then either be in the context of the call on the left, which includes lock passing, or independent. The first would use the dot, the second an exclamation mark.

5.10.6 Exceptions and Compensations

As mentioned in section 5.4, compensations can be seen as a generalization of exceptions. We drafted a mechanism to use compensations as a replacement or an alternative to exceptions and, in the future, would like to explore this possibility.

5.10.7 Security

The basic design of D-Scoop avoids some security issues: by leveraging information of the garbage collection, only clients that received a reference from a trusted source can issue calls to the associated object. This is complemented by not allowing the creation of remote object except through application-defined factories. However, the current prototype does not yet include more advanced security mechanisms such as encryption and certificates, all of which are an important measure when D-Scoop is used outside of a trusted network. Since the focus of this work is on showing the distributed application of the Scoop model, we leave the security considerations as future work.

5.11 Conclusion

This chapter presented D-Scoop, a distributed programming model obtained by combining the network objects abstraction with the runtime semantics of the object-oriented concurrency model Scoop. We presented an efficient two-phase locking protocol that generalized the strong reasoning guarantees of Scoop to network objects, allowing for interference-free and transaction-like reasoning on (potentially multiple) remotely located objects, without the programmer having to explicitly manage their synchronization. Furthermore, we proposed a compensation mechanism by which D-Scoop programs can recover from failure. The evaluation of our prototype implementation [18] suggested that D-Scoop remains competitive against—and can outperform—explicit locking-based synchronization in Java RMI, a well-established realization of network objects, with the automatic synchronization mechanisms also allowing for more compact code.

Chapter 6

D-Scoop Semantics

6.1 Introduction

The previous chapters introduced Scoop and the basic extensions for distribution in an informal way. The focus was on the perspective of a developer using D-Scoop and the general architecture of the system. What remains is to explain how D-Scoop works internally, which is the purpose of this chapter.

While it is possible to specify a protocol by giving specifications for every message type, it becomes difficult to describe dependencies between messages if the protocol is partly asynchronous. For example, the IMAP protocol [15] allows parallel processing of commands, if no ambiguity exists. A reason for the first revision of the IMAP 4 specification was the problem of identifying ambiguity, for which IMAP4rev1 gives a few examples. The issue can be avoided completely by giving an abstract formal specification defining what messages can be sent in parallel in what situation.

The following sections describe the semantics of D-Scoop independently of the underlying programming language. This is achieved through abstraction of the state of a processor: classes, objects and their state are not modeled. In consequence, the effects of the application of a call are non-deterministic, similar to internal choices in CSP [30]. Internal operations that only change the state of objects are completely omitted. Since only further calls have an effect on the abstract state of a processor, the effects of the application of a call are limited to further calls and the registration of compensation calls. Without an underlying programming language, there is no abstract syntax tree. Therefore, the semantics does not take the form of a regular operational semantics [56].

Aside from a specification of the internal D-Scoop system, the semantics can be used to show that a specific exchange of messages is valid. From this perspective, the D-Scoop semantics is a form of context-sensitive grammar for a language composed of messages on a timeline.

The state of a D-Scoop system, even without taking into account objects, is fairly complex. We developed a system based on the concept of *cells*, together with lists and sets to structure the state of the system in a compact, yet intuitive from. The necessary formal rules are described in section 6.3. Section 6.4 presents the structure of the state. The rules for state transitions are covered in sections 6.5 to 6.10. These sections start with the basic message passing system, handle locking and execution, and finally end with describing failure cases using exceptions and compensation.

$$
\begin{array}{rcll}
i & \in & \mathbb{N} & \text{Value} \\
cell & ::= & \langle c* \rangle & \text{Cell} \\
c & ::= & cell \mid i \mid \epsilon & \text{Cell Content} \\
[L] & ::= & [l(L)] & \text{List} \\
l(L) & ::= & L \mid l(L); l(L) \mid \epsilon & \text{List Content}
\end{array}
$$

Figure 6.1: Syntax for cells and lists

The sections contain small examples; two large examples are presented in chapter 4, which also makes a connection between concrete Eiffel D-Scoop code, exchanged messages and the semantics presented here.

6.2 Semantics and Implementation

The actual implementation in form of a prototype based on EiffelStudio differs slightly due to the fact that multiple processors share a node. This situation is used in the prototype to optimize communication between local processors. Rather than sending messages, these processor directly manipulate the other processor's state while using locks to avoid data races. Nevertheless, the effects are the same and local processors are in no way treated preferably to remote processors: the usage of shared memory is limited to optimization without impact on the semantics.

Also, if messages, such as PRELOCK or LOCK, are sent to multiple processor residing on the same node, they are merged into one message. We acknowledge that the semantics of the SHARE and RELEASE messages cannot be described using the abstraction given in this chapter, but we refer to [5] for the description of the garbage collection mechanism that these two message types are used for.

After reading through the semantics, the reader will notice that processors often have a non-deterministic choice between acting on a message in their in-box, handling prelock or executing an instruction. In practice, these things can be handled by two threads in parallel. The role of threads[1] behind the proxy processors is to handle the in-boxes and prelock queues of all the suppliers of the client it is proxy of. This design allows for a seamless integration of D-Scoop into Scoop without preferential treatment of local processors and no performance regression if no remote processor is involved. Furthermore, it also allows the node to only allocate as much proxy processors, and thereby threads, as currently needed.

6.3 Definitions and Support

In order to structure the state of a D-Scoop system in a compact and understandable way, we need to introduce some concepts. Although a complex state can be structured using abstract data types [39], we chose to introduce a system of cells, lists and sets which offers a more intuitive representation.

[1] we use the term thread here as this is how processors in Scoop and D-Scoop are based upon

6.3.1 Cells

Informal definition

Our goal is to make the inference rules as concise as possible in order to make the semantics easily understandable. Since the state of a D-Scoop system is complexScoopA stated design goal of Scoop was that much of the complexity of writing concurrent systems should be part of the runtime environment and the compiler instead of the program. This carries over to D-Scoop., we needed a method to abstract away state that is not relevant to the given rule. For this, we introduce *cells*, which is a concept inspired by the K framework [28]. A cell has a *label* for identification and can contain other cells or values. We use angular brackets to frame the cell. A concrete cell with two inner cells of which one contains two more may look like this:

$$\langle\langle 5\rangle_{\mathrm{B}}\langle\langle 4\rangle_{\mathrm{X}}\langle 4\rangle_{\mathrm{Y}}\rangle_{\mathrm{C}}\rangle_{\mathrm{A}}$$

In contrast to tuples, the order of the contents in a cell has no significance, making cells similar to sets. For clarity, a cell contains either a single value, such as a number or a tuple, or a various number of cells, but never multiple values or a mix of values and cells. The syntax of cells is shown in fig. 6.1.

Informally, we can say that for applying a rule that has a cell A in its conclusion, this rule can be applied using an effective cell B if every component of A matches a distinct component in B with the same label. Note that this relationship is not symmetric: A can match B even if B has some components not in A: this is intentional to omit components in rules that have no effect on whether the rule can be used or not. If there is a component in A with a label that no component in B shares, the two cells are not matching. For example, the following inference rule would match the cell given above:

$$\frac{x > 2}{\langle\langle x\rangle_{\mathrm{B}}\langle\langle y\rangle_{\mathrm{Y}}\rangle_{\mathrm{C}}\rangle_{\mathrm{A}} \to \langle\langle x\rangle_{\mathrm{B}}\langle\langle y+1\rangle_{\mathrm{Y}}\rangle_{\mathrm{C}}\rangle_{\mathrm{A}}}\ \textsc{example rule 1}$$

Whereas, since X is not a tag of the outer cell, the following one would not match:

$$\frac{}{\langle\langle\langle y\rangle_{\mathrm{Y}}\rangle_{\mathrm{X}}\rangle_{\mathrm{A}} \to \langle\langle\langle y+1\rangle_{\mathrm{Y}}\rangle_{\mathrm{X}}\rangle_{\mathrm{A}}}\ \textsc{example rule 2}$$

Also, components of a cell not listed on the right hand side of an inference rule are not changed if the rule is applied. Therefore given the cell above and the rule example rule 1, the resulting cell would be:

$$\langle\langle 5\rangle_{\mathrm{B}}\langle\langle 4\rangle_{\mathrm{X}}\langle 5\rangle_{\mathrm{Y}}\rangle_{\mathrm{C}}\rangle_{\mathrm{A}}$$

In addition, if a rule does not need all the context, it is possible to leave out the surrounding cells. If we simplify matching accordingly, we can write it more concisely by removing the surrounding A-cell:

$$\frac{x > 2}{\langle x\rangle_{\mathrm{B}}\langle\langle y\rangle_{\mathrm{Y}}\rangle_{\mathrm{C}} \to \langle x\rangle_{\mathrm{B}}\langle\langle y+1\rangle_{\mathrm{Y}}\rangle_{\mathrm{C}}}$$

Of course, this simplification might make the rule apply in more situations than intended.

Another simplification allows for constant cells to be mentioned only once. In the above rule, the cell B is mentioned so that the $x > 2$ condition can be checked. The cell does not change, so we have to mention it only once. We use the $\vdash$ symbol to delimit the common part from the changing part.

$$\frac{x > 2}{\langle x \rangle_{\mathrm{B}} \vdash \langle\langle y \rangle_{\mathrm{Y}} \rangle_{\mathrm{C}} \to \langle\langle y + 1 \rangle_{\mathrm{Y}} \rangle_{\mathrm{C}}}$$

We can remove cell C to get a very concise rule, but note that the rule can now be applied in more cases:

$$\frac{x > 2}{\langle x \rangle_{\mathrm{B}} \vdash \langle y \rangle_{\mathrm{Y}} \to \langle y + 1 \rangle_{\mathrm{Y}}}$$

Formal Definition

Figure 6.2 shows the inference rules that realize the mechanisms given above. The rules TAKE OUT 1 and TAKE OUT 2 are used to separate a cell common to both left and right. The rules STRIP 1 and STRIP 2 are used to remove outer cells, but only apply if there is a single cell in the non-common part of the rule. This is to reduce ambiguity.

In order to mention only the components the rule affects or requires, the other components are hidden. This is a three step process: first, the rule EXTRACT 1 is used to extract one of the components of the cell. This can be repeated with EXTRACT 2 until the cell with the components we want to hide is on the left, with a series of cells delimited by $\ltimes$ on the right. The $\ltimes$ symbol represents insertion: The cells on the left are put into the cell on the right. The CELL HIDE rule then removes one of the components of the cell. The rules INSERT 1 and INSERT 2 re-pack the cells until there is no $\ltimes$ symbol.

The ϵ symbol is a neutral element within cells and symbolizes no component. It allows for partitioning of cells with just one component, e. g.: $\langle 5 \rangle_{\mathrm{A}} = \langle XY \rangle_{\mathrm{A}}, X = \epsilon, Y = 5$.

In general, a variable in a cell may stand for multiple cells, not just a single one. So $\langle\langle\rangle_{\mathrm{B}}\langle\rangle_{\mathrm{C}}\rangle_{\mathrm{A}}$ may be matched by $\langle XY \rangle_{\mathrm{A}}$ with either $X = \langle\rangle_{\mathrm{B}}, Y = \langle\rangle_{\mathrm{C}}$ or $X = \langle\rangle_{\mathrm{B}}\langle\rangle_{\mathrm{C}}, Y = \epsilon$ or $X = \epsilon, Y = \langle\rangle_{\mathrm{B}}\langle\rangle_{\mathrm{C}}$.

Limitation. Due to the possibility to hide cells, a rule cannot check for the absence of a specific cell within a cell: This rule would match even if the cell exists, as the cell could simply be hidden. This limitation does not apply if a cell is in a set or a list. Lists and sets are explained below.

Cells can be added or removed. Hiding of content is not possible in either case, since only content that is the same on both sides of the transition is allowed. This is intuitive for adding of cells, but for removal it is not, at least at first. However, removal of cells is like adding of cells if a rule is applied in reverse.[2]

[2] A careful reader will notice that some of the rules introduced later are not adhering to this: for sake of readability and brevity, we ommitted some cell content during removal where it is clear which cell is removed from the context.

$$\frac{\langle A\rangle_a \vdash B \to B'}{\langle A\rangle_a B \to \langle A\rangle_a B'}\ \textsc{take out 1}$$

$$\frac{\langle A\rangle_a\langle B\rangle_b \vdash C \to C'}{\langle A\rangle_a \vdash \langle B\rangle_b C \to \langle B\rangle_b C'}\ \textsc{take out 2}$$

$$\frac{A \to A'}{\langle A\rangle_l \to \langle A'\rangle_l}\ \textsc{strip 1}$$

$$\frac{A \vdash B \to A \vdash B'}{A \vdash \langle B\rangle_l \to A \vdash \langle B'\rangle_l}\ \textsc{strip 2}$$

$$\frac{A \ltimes \langle B\rangle_l \to A' \ltimes \langle B'\rangle_l}{\langle AB\rangle_l \to \langle A'B'\rangle_l}\ \textsc{extract 1}$$

$$\frac{A \ltimes \langle B\rangle_l \ltimes X \to A' \ltimes \langle B'\rangle_l \ltimes X'}{\langle A\ B\rangle_l \ltimes X \to \langle A'\ B'\rangle_l \ltimes X'}\ \textsc{extract 2}$$

$$\frac{\langle B\rangle_l \ltimes X \to \langle B'\rangle_l \ltimes X'}{\langle A\ B\rangle_l \ltimes X \to \langle A\ B'\rangle_l \ltimes X'}\ \textsc{cell hide}$$

$$\frac{\langle A\ B\rangle_l \ltimes X \to \langle A'\ B'\rangle_l \ltimes X'}{A \ltimes \langle B\rangle_l \ltimes X \to A' \ltimes \langle B'\rangle_l \ltimes X'}\ \textsc{insert 1}$$

$$\frac{\langle A\ B\rangle_l \ltimes X \to \langle A'\ B'\rangle_l \ltimes X'}{A \ltimes \langle B\rangle_l \ltimes X \to A' \ltimes \langle B'\rangle_l \ltimes X'}\ \textsc{insert 2}$$

Figure 6.2: Rules for simplifying cells

6.3.2 Sets and lists

In addition to cells, our notation also employs sets and lists. Sets use curly braces and a comma for delimiting its entries. Sets can be empty, for which we use the normal symbol for an empty set, $\emptyset$. We only use sets of tuples or scalar values, so we omit inference rules and simply use standard set operations, like for example the union operator $\cup$, which is generally used for partitioning sets in rules. There is no rule that allows parts of a set be hidden, otherwise the $\notin$ operator would become meaningless.

Lists are written between brackets and use semicolons as delimiter. The semicolon is an associative operator, enabling matching at the front ($[a; q]$), the end ($[q; a]$) and somewhere in the middle ($[q_1; a; q_2]$) of the list. If used as a queue, new entries are added from the right as a convention, and taken out from the left. As a stack, the items are added and removed from the left. Empty lists is written as $[\epsilon]$. The epsilon symbol ϵ is also the neutral element of the semicolon operator, so that the following holds:

$$[a; b] = [a; \epsilon; b]$$
$$[a] = [\epsilon; a] = [a; \epsilon]$$

The usage of a neutral element enables rules like:

$$\frac{a \in \mathbb{N}}{[a; q] \to [q]}$$

to match lists with a single item like $[15]$, which would be reduced to $[\epsilon]$ by this rule.

Hiding in lists

Informal Definition When dealing with lists, ellipses can be used to hide parts of lists. For example, the following rule removes an integer value from a list in cell C if it is on the left of the list and greater than five:

$$\frac{a > 5}{\langle [x...] \rangle_{\mathrm{C}} \to \langle [...] \rangle_{\mathrm{C}}}$$

We mentioned before that cells either contain an arbitrary amount of other cells or a single value. The latter includes lists and sets. For readability and convenience, if a cell contains a list, the angular brackets of the cells are omitted, so only the brackets of the list remain. The same is done for a set in a cell, where the curly braces remain. By applying this definition given in fig. 6.4 to the inference rule above, we derive the following rule:

$$\frac{a > 5}{[x...]_{\mathrm{C}} \to [...]_{\mathrm{C}}}$$

$$\frac{A \ltimes [H; \diamond; T] \ltimes X \to A' \ltimes [H; \diamond; T] \ltimes X' \qquad \diamond \notin [H; A; T]}{[H; A; T] \ltimes X \to [H; A'; T] \ltimes X'} \text{ LIST EXTRACT}$$

$$\frac{[H; A; T] \ltimes X \to [H; A'; T] \ltimes X'}{A \ltimes [H; \diamond; T] \ltimes X \to A' \ltimes [H; \diamond; T] \ltimes X'} \text{ LIST INSERT}$$

$$\frac{[A \ldots] \ltimes C \to [A' \ldots] \ltimes C'}{[A; T] \ltimes C \to [A'; T] \ltimes C'} \text{ LIST HIDE 1}$$

$$\frac{[\ldots A] \ltimes C \to [\ldots A'] \ltimes C'}{[H; A] \ltimes C \to [H; A'] \ltimes C'} \text{ LIST HIDE 2}$$

$$\frac{[\ldots A \ldots] \ltimes C \to [\ldots A' \ldots] \ltimes C'}{[H; A; T] \ltimes C \to [H; A'; T] \ltimes C'} \text{ LIST HIDE 3}$$

$$\frac{[\ldots] \ltimes C \to [A' \ldots] \ltimes C'}{[T] \ltimes C \to [A'; T] \ltimes C'} \text{ LIST PREPEND}$$

$$\frac{[\ldots] \ltimes C \to [\ldots A'] \ltimes C'}{[T] \ltimes C \to [T; A'] \ltimes C'} \text{ LIST APPEND}$$

$$\frac{[A \ldots] \ltimes C \to [\ldots] \ltimes C'}{[A; T] \ltimes C \to [T] \ltimes C'} \text{ LIST CHOP LEFT}$$

$$\frac{[\ldots A] \ltimes C \to [\ldots] \ltimes C'}{[H; A] \ltimes C \to [H] \ltimes C'} \text{ LIST CHOP RIGHT}$$

$$\frac{[\ldots A \ldots] \ltimes C \to [\ldots] \ltimes C'}{[H; A; T] \ltimes C \to [H; T] \ltimes C'} \text{ LIST REMOVE}$$

Figure 6.3: Rules for simplifying lists

$$
\begin{aligned}
\langle A\ B\rangle_{lbl} &= \langle B\ A\rangle_{lbl} \\
\langle A\ \epsilon\rangle_{lbl} &= \langle A\rangle_{lbl} \\
[A;\epsilon] &= [A] = [\epsilon;A] \\
[A;\epsilon;B] &= [A;B] \\
\langle [A]\rangle_l &= \big[A\big]_l \\
\langle \{A\}\rangle_l &= \big\{A\big\}_l
\end{aligned}
$$

Figure 6.4: Definitions for cells and lists

Formal definition. Figure 6.3 shows the rules that deal with simplification of lists. The rule LIST EXTRACT works similar to its counterparts for cells, EXTRACT 1 and EXTRACT 2. It allows an item to be extracted out of a list so that some of it can be hidden using another rule before it is re-inserted into the same position by LIST INSERT. It uses the $\diamond$ symbol is a placeholder so that the modified sub-list is inserted at the right spot.

Apart from the already discussed CELL HIDE rule, there are many rules to hide parts of a list. The three LIST HIDE rules hide the head, the tail or both from a list by replacing them with an ellipsis. LIST PREPEND allows a variable number of new items denoted by A' to be added on the left of a list, whereas LIST APPEND does the same on the right. The two rules LIST CHOP LEFT and LIST CHOP RIGHT are the counterparts to the rules mentioned before: instead of adding, they remove items from the left or right of the list. Finally, LIST REMOVE can be used to remove a part from somewhere in the list.

6.3.3 Definitions

There are multiple ways to write down a set, list or cell. To describe the equivalent notations, we use a small set of definitions for cells, lists and sets shown in fig. 6.4. The first definition makes the ordering of components in cells irrelevant, that is, the implicit delimiter of components in cells is commutative. The second rule introduces ϵ as a neutral element.

The next two definitions establish that a list with an ϵ at the left or right of a list is equal to the list without ϵ. The other rule establishes that an ϵ in the middle of a list is also irrelevant. These two rules together introduce ϵ as a neutral element in lists.

The last two definitions introduce a short-hand notation of lists and sets in cells, as mentioned in the previous section.

6.4 Configuration

A configuration of the D-SCOOP semantics consists of a set of processors and its syntax is described in fig. 6.5. For the purpose of the semantics, every processor is considered to be on a separate node. While in practice a node may hold multiple processors[3], for our abstraction this makes no difference since semantically, D-SCOOP does not differentiate between local and remote processors. The messages that have no meaning

[3]In fact, new processors are created on the same node as the creating processor.

$$
\begin{array}{rcll}
p & \in & \mathbb{N} & \text{Proc. Id} \\
c & \in & \mathbb{N} & \text{Client Id} \\
s & \in & \mathbb{N} & \text{Supplier Id} \\
f & \in & \mathbb{N} & \text{Frame Id} \\
m & \in & \mathbb{N} & \text{Message Id} \\
S & ::= & \left\langle \{p\}_{\text{Ps}} \, P* \right\rangle_{\text{C}} & \text{State} \\
P & ::= & \left\langle \begin{array}{l} \langle p \rangle_{\text{Pi}} \, [F]_{\text{El}} \, [R]_{\text{Pl}} \, \{N\}_{\text{Ns}} \, \langle W \rangle_{\text{W}} \\ \quad [M_m]_{\text{Ib}} \, [M_m \uparrow p]_{\text{Ob}} \end{array} \right\rangle_{\text{P}} & \text{Processor} \\
F & ::= & \left\langle \langle f \rangle_{\text{Fi}} \langle T \rangle_{\text{T}} \, [I]_{\text{Il}} \, [p]_{\text{Ol}} \, [C]_{\text{Cl}} \, \{L\}_{\text{As}} \, \{L\}_{\text{Is}} \right\rangle_{\text{F}} & \text{Frame} \\
R & ::= & (c, m) \mid c & \text{Prelock} \\
N & ::= & (c, m) & \text{Subscriptions} \\
W & ::= & \mathfrak{N} \mid \mathfrak{R} \mid \mathfrak{W}_m & \text{Wait State} \\
T & ::= & \mathfrak{L} \mid \mathfrak{C} \mid \mathfrak{S} & \text{Frame Type} \\
L & ::= & (s, f) & \text{Lock} \\
I & ::= & \pi_c \mid \upsilon \mid \xi \mid \Omega \mid m \downarrow s \mid \downarrow V & \\
 & \mid & \textsc{ACall}\ s, \{s\} \mid \textsc{SCall}\ s, \{s\} \mid \textsc{Pass}\ \{L\}, s & \\
 & \mid & \textsc{Exec}\ \{s\}, \{L\}, \{L\}, \{s\} \mid \textsc{Notify}\ \{W\} & \\
 & \mid & \textsc{Unlock}\ \{s\} \mid \textsc{Return}\ c, f \mid \textsc{Await}\ \{s\} & \\
 & \mid & \textsc{Revert} \mid \textsc{Retract}\ \{s\} & \text{Instructions} \\
M & ::= & \boxed{\text{Prelock } c} \mid \boxed{\text{Lock } c} \mid \boxed{\text{Pass } c, f, p} \mid \boxed{\text{Unlock } c, f} & \\
 & \mid & \boxed{\text{ACall } c, f, \{s\}} \mid \boxed{\text{SCall } c, f, \{s\}, \{L\}} & \\
 & \mid & \boxed{\text{Retract } c} \mid \boxed{\text{Revert } c, s} \mid \boxed{\text{Await } c} & \\
 & \mid & \boxed{\text{OK}} \mid \boxed{\text{OK } L} \mid \boxed{\text{FAIL}} & \text{Messages} \\
C & ::= & \left(\left[\{s\} \right], c \right) & \text{Compensation}
\end{array}
$$

Figure 6.5: Syntax of D-Scoop configuration

in this abstract semantics are Hello, Ping, Index, Share and Release. While the first two are only used for technical reasons, the latter three are meaningless in the abstract semantics for another reason: the described semantics has no concepts of objects, values and classes. As a consequence, we do not need to model the heap, the type system and all operations except calls. This is possible since the locking mechanism is only dependent on the processors involved in a call, not the actual objects. In addition, this generalizes the semantics to apply to non-object-oriented programming languages, or languages that use a different object system. Calls can therefore be modeled simply by a client processor, a supplier processor, and a possibly empty set of argument processors. Also, modeling the type system is redundant as it is already described in detail in section 3.1.1 and in previous publications such as [49].

As a convention, the second letter in the label indicates an identifier (I), a set (S) or a list (L). The only exceptions are the in- and out-box for messages.

A processor consists of the following:

1. the identifier of the processor PI
2. the execution list EL
3. the prelock list PL
4. the message in-box (a list) IB
5. the message out-box (also a list) OB
6. and the notification set NS

Every processor has a globally unique identifier (PI). The execution list is used as both a stack (in case of synchronous calls) and as a queue (for incoming lock requests). The prelock list is used as a stack. The message lists are used as queues. Queues are processed from front (left) to back (right), with new items appended at the back. These lists can often make progress independently from each other, yielding some intra-processor parallelism. This is exploited by the implementation as explained in section 6.2 using proxy processors.

Execution list. The execution list (EL) contains the execution frames[4] (F) and can be considered the core of the processor. Only instructions from the active frame are processed, which is always the leftmost in the execution list. Note that the locking token for receiving messages is not considered an instruction and can be removed even if it is not in the active frame[5]. Frames are added to the execution list from both the left and the right, depending on the circumstances.

Each frame has a unique identifier (FI). The most important component of a frame is the instruction list (IL), which contains the instructions for the processor to execute. Instructions can change the state of the current processor, including putting messages in the out-box. No instruction accesses the state of other processors directly. Execution of an instruction may replace the instruction with one or more further instructions. Incoming requests from other processors are always appended to the right of the instruction list, while only the instructions on the left are processed. Details of the various instructions are explained in the following sections.

The frame also contains the owners list (OL), which is used as a stack with the current owner on top, that is, on the left. The compensation and client list (CL) is

[4]While the implementation presented before uses the terms queue of queue and subqueue as a reference to West [66], these semantics merge this concept into the execution frames: there is no reason to keep the two apart, it only adds complexity.

[5]The token can be seen as a variable that receives a value once the matching message arrives, and the position of the token represents the point where the processor waits until this variable has received a value.

$$\langle\langle\emptyset\rangle_{\text{Ps}}\rangle_{\text{C}}$$

Figure 6.6: Initial D-Scoop configuration

used for rollbacks in case of failure, and is discussed in section 6.10.2. Finally, the two lock sets As and Is track the locks currently held by the processor while this frame is active. The first tracks the acquired locks, that is, the lock acquired through locking, while the second tracks the locks inherited through separate or non-separate lock passing.

Prelock list. The prelock list (Pl) contains information about the current prelock state of the processor and is required for the prelock protocol as introduced in section 5.3. Section 6.7.1 gives more detail about the exact usage of this data structure.

Message lists. The message lists are used for sending and receiving messages. They are used as queues and discussed in section 6.5.

$$\text{suppliers}\,(L) = \{s \mid \exists i : (s, i) \in L\} \tag{6.1}$$

Extracting Suppliers

The function suppliers (L) defined through eq. (6.1) is used to transform a set of locks into a set that only contains the supplier identifiers. This is used to compare the set of required locks with the set of locks already acquired: the former consists of scalar values, while the other contains tuples.

6.4.1 Initial Configuration

Initially, a D-Scoop system contains just the set of processor identifiers. This configuration is shown in fig. 6.6. Processors can appear in the system via the rule appear presented in section 6.6.

6.5 Message Passing

As mentioned in section 5.2, D-Scoop is built on top of a simple asynchronous message passing system: messages are divided into requests and replies. Every message has a subject, in the case of replies it is either [OK] or [FAIL]. Under normal circumstances, all replies are [OK]; the usage of [FAIL] is discussed in section 6.10. Sending a message is done by placing it in the out-box of the sending processor. As a separate step, the message is put into the in-box of the receiving processor by transmit, which is the only rule apart from spawn in the semantics that involves two processors. If it is an expected reply, the rules recv. ok and recv. value are handling the message from there, by removing or replacing the blocking $\downarrow$ symbol. Handling requests is done by matching and removing the first message directly in the associated inference rule.

$$\frac{}{\left\langle \left[m_i \uparrow r \ldots \right]_{\text{OB}} \right\rangle_{\text{P}} \left\langle \left\langle r \right\rangle_{\text{PI}} \left[\ldots \right]_{\text{IB}} \right\rangle_{\text{P}} \rightarrow \left\langle \left[\ldots \right]_{\text{OB}} \right\rangle_{\text{P}} \left\langle \left\langle r \right\rangle_{\text{PI}} \left[\ldots m_i \right]_{\text{IL}} \right\rangle_{\text{P}}} \text{ TRANSMIT}$$

$$\frac{}{\begin{array}{c}\left[\ldots \left\langle \left[\ldots i \downarrow s \ldots \right]_{\text{IL}} \right\rangle_{\text{F}} \ldots \right]_{\text{EL}} \left[\boxed{\text{OK}}_i \ldots \right]_{\text{IB}} \\ \rightarrow \left[\ldots \left\langle \left[\ldots \epsilon \ldots \right]_{\text{IL}} \right\rangle_{\text{F}} \ldots \right]_{\text{EL}} \left[\ldots \right]_{\text{IB}}\end{array}} \text{ RECV. OK}$$

$$\frac{}{\begin{array}{c}\left[\ldots \left\langle \left[\ldots i \downarrow s \ldots \right]_{\text{IL}} \right\rangle_{\text{F}} \ldots \right]_{\text{EL}} \left[\boxed{\text{OK v}}_i \ldots \right]_{\text{IB}} \\ \rightarrow \left[\ldots \left\langle \left[\ldots \downarrow v \ldots \right]_{\text{IL}} \right\rangle_{\text{F}} \ldots \right]_{\text{EL}} \left[\ldots \right]_{\text{IB}}\end{array}} \text{ RECV. VALUE}$$

Figure 6.7: D-Scoop messaging rules

$$\frac{\text{fresh}\,(f) \wedge o \notin P \wedge p \notin P}{\begin{array}{c}\left\langle P \right\rangle_{\text{PS}} \rightarrow \\ \left\langle P \cup \{p\} \right\rangle_{\text{PS}} \\ \left\langle p \right\rangle_{\text{PI}} \\ \left[\left\langle \left\langle f \right\rangle_{\text{FI}} \left\langle \mathfrak{S} \right\rangle_{\text{T}} \left[\Omega; v \right]_{\text{IL}} \left[\epsilon \right]_{\text{OL}} \left[\left(\left[\epsilon \right], o \right) \right]_{\text{CL}} \left\langle \emptyset \right\rangle_{\text{AS}} \left\langle \emptyset \right\rangle_{\text{IS}} \right\rangle_{\text{F}} \right]_{\text{EL}} \\ \left[\epsilon \right]_{\text{PL}} \left[\epsilon \right]_{\text{IB}} \left[\epsilon \right]_{\text{OB}} \left\langle \emptyset \right\rangle_{\text{NS}}\end{array}\Bigg]_{\text{P}}} \text{ APPEAR}$$

$$\frac{\text{fresh}\,(p)}{\begin{array}{c}\left\langle P \right\rangle_{\text{PS}} \left[\left[\left\langle \left[\textsc{Spawn} \ldots \right]_{\text{IL}} \right\rangle_{\text{F}} \ldots \right]_{\text{EL}} \right]_{\text{P}} \\ \rightarrow \\ \left\langle P \cup \{p\} \right\rangle_{\text{PS}} \left[\left[\left\langle \left[\ldots \right]_{\text{IL}} \right\rangle_{\text{F}} \ldots \right]_{\text{EL}} \right]_{\text{P}} \\ \left[\left\langle p \right\rangle_{\text{PI}} \left[\epsilon \right]_{\text{EL}} \left[\epsilon \right]_{\text{PL}} \left[\epsilon \right]_{\text{IB}} \left[\epsilon \right]_{\text{OB}} \left\langle \emptyset \right\rangle_{\text{NS}} \right]_{\text{P}}\end{array}} \text{ SPAWN}$$

Figure 6.8: Inference rules for appearance and spawning of processors

6.6 Appearance

The SPAWN and APPEAR rules shown in fig. 6.8 govern the addition of processors to the system.

New Processors can *appear* in the D-Scoop system. This is due to an external event, for example if a person starts a D-Scoop program and connects it to the system. This spontaneous appearance is governed by the APPEAR rule. An appearing processor has always some initialization code, represented by an initial frame with an Ω. Note that the compensation list contains a client identifier that is not used by any processors in the system. This indicates that the reason for executing this frame is external. It could be due to a processor that joins the system by additional applications of APPEAR

Creation of new processors is also caused by the Spawn instruction and the SPAWN rule. In our implementation, SPAWN corresponds to the creation of a processor using **`create`**. It creates a new processor cell. The processor has a new, unique identifier, which is also added to the set of identifiers in the Ps cell. None of the lists and sets in the new processor contain any items.

Note that it is up to the client to gain a lock on the new processor in order to issue any creation procedures for new objects. Without a representation of objects, it is not

possible to give a rule when a processor can be discarded, as we do not track which processors have references to each other.

6.7 Lock Handling

The handling of locks is the central aspect of D-Scoop, and it is the differentiating feature of D-Scoop compared to other distributed active object models. It naturally occupies the majority of the semantics, and part of its complexity in D-Scoop is due to the presence of failure and the compensation mechanism. The latter interacts in a complex way with the concept of *lock passing*, which not only gives processors the ability to pass locks, but also obliges them to take responsibility for everything the suppliers do with the locks.

6.7.1 Acquiring Locks

Lock

A lock is a combination of a processor identifier and a frame identifier, written as a tuple. The frame identifier is needed for sending requests to the correct frame, as it is possible that the same processor has multiple locks on the same other processor through repeated lock passing, but on different frames.

The exec instruction

Before the body of a call can be executed, the required processor locks need to be acquired. The Exec R,O,P instruction is taking care of acquiring the locks. The three arguments of this instruction are:

Required locks. The first argument of this instruction is the set of required locks. This is the set of processor identifiers from the arguments of the call that are not yet locked due to lock passing. The current processor is implicitly locked, so this set does not include it. Unlike implied by its name, the set only contains identifiers, not lock tuples, since the frame identifier is not yet known.

Obtained locks. The second argument is the set of obtained locks. These are the locks that have been passed from another processor through a separate synchronous call. In the case of non-separate or asynchronous calls, this set is always empty, whereas in the other case it will contain at least a lock on the client.

Prelocked processors. The last argument is used to identify the processors where the [Prelock] message has already been sent. It contains only processor identifiers.

Rules

Figure 6.9 contains the rules for acquiring locks. At first, unless there is no lock to be acquired, the only rule that can be applied is the prelock rule. It appends to the last argument, the prelocked processors, of the Exec instruction with the processor with the lowest identifier. But it also sends a [Prelock] message and adds a receive token in front, so that the instruction queue is blocked until the appropriate response has been received. Since the messages are sent in a global order based on the identifiers, the

$$\frac{\begin{array}{c} R \setminus P = \{s\} \cup X \\ \forall x \in X : s < x \\ \text{fresh}\,(i) \end{array}}{\begin{array}{c} \langle c \rangle_{\text{PI}} \vdash \\ \left[\langle \left[\textsc{Exec}\ R, O, P \ldots\right]_{\text{IL}} \langle A \rangle_{\text{AS}} \langle I \rangle_{\text{IS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \\ \left[\ldots\right]_{\text{OB}} \\ \rightarrow \\ \left[\langle \left[i \downarrow s; \textsc{Exec}\ R, O, P \cup s \ldots\right]_{\text{IL}} \langle A \rangle_{\text{AS}} \langle I \rangle_{\text{IS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \\ \left[\ldots \boxed{\text{Prelock } c}_i \uparrow s\right]_{\text{OB}} \end{array}}\ \textsc{prelock}$$

$$\frac{\begin{array}{c} R \setminus \text{suppliers}\,(A \cup O) = \{s\} \cup X \\ \text{fresh}\,(i) \end{array}}{\begin{array}{c} \langle c \rangle_{\text{PI}} \vdash \left[\langle \left[\textsc{Exec}\ R, O, R \ldots\right]_{\text{IL}} \langle A \rangle_{\text{AS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \left[\ldots\right]_{\text{OB}} \rightarrow \\ \left[\langle \left[i \downarrow s; \textsc{Exec}\ R, O, R \ldots\right]_{\text{IL}} \langle A \rangle_{\text{AS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \left[\ldots \boxed{\text{Lock } c}_i \uparrow s\right]_{\text{OB}} \end{array}}\ \textsc{lock}$$

$$\frac{}{\begin{array}{c} \left[\langle \left[\downarrow l; \textsc{Exec}\ R, O, R \ldots\right]_{\text{IL}} \langle A \rangle_{\text{AS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \rightarrow \\ \left[\langle \left[\textsc{Exec}\ R, O, R \ldots\right]_{\text{IL}} \langle \{l\} \cup A \rangle_{\text{AS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \end{array}}\ \textsc{add lock}$$

$$\frac{}{\begin{array}{c} \left[\ldots\right]_{\text{PL}} \left[\boxed{\text{Prelock } c}_i \ldots\right]_{\text{IB}} \langle N \rangle_{\text{NS}} \rightarrow \\ \left[\ldots\,(c, i)\right]_{\text{PL}} \left[\ldots\right]_{\text{IB}} \langle N \setminus \{c\} \rangle_{\text{NS}} \end{array}}\ \textsc{prelock enqueue}$$

$$\frac{}{\left[(c, i) \ldots\right]_{\text{PL}} \left[\ldots\right]_{\text{OB}} \rightarrow \left[c \ldots\right]_{\text{PL}} \left[\ldots \boxed{\text{OK } s}_i \uparrow c\right]_{\text{OB}}}\ \textsc{prelock reply}$$

$$\frac{\text{fresh}\,(f)}{\begin{array}{c} \langle s \rangle_{\text{PI}} \vdash \\ \left[\ldots\right]_{\text{EL}} \left[c \ldots\right]_{\text{PL}} \left[\boxed{\text{Lock } c}_i \ldots\right]_{\text{IB}} \left[\ldots\right]_{\text{OB}} \\ \rightarrow \\ \left[\begin{array}{c} \ldots \\ \langle \langle f \rangle_{\text{FI}} \langle \mathfrak{S} \rangle_{\text{T}} \left[\epsilon\right]_{\text{IL}} \left[c\right]_{\text{OL}} \left[\left[\left(\left[\epsilon\right], c\right)\right]_{\text{CL}}\right]_{\text{CL}} \langle \emptyset \rangle_{\text{AS}} \langle \emptyset \rangle_{\text{IS}} \rangle_{\text{F}} \end{array}\right]_{\text{EL}} \\ \left[\ldots\right]_{\text{PL}} \left[\ldots\right]_{\text{IB}} \left[\ldots \boxed{\text{OK } s, f}_i \uparrow c\right]_{\text{OB}} \end{array}}\ \textsc{lock reply}$$

Figure 6.9: Inference rules for locking

prelock protocol does not deadlock. In an implementation with nodes, such as our prototype, the processor identifier should consist of a globally unique node identifier plus a (locally) unique processor identifier. The ordering would then be first by node identifier, then by processor identifier. With this identification scheme, the prelock requests for processors residing on the same node can be merged into one message.

On the supplier side, the prelock message is put into the prelock queue according to PRELOCK ENQUEUE). The prelock queue contains either scalar identifier values, or tuples of identifiers. The latter contains the message id of the original prelock request alongside the client identifier, whereas the former is only the client identifier. Sending of the reply is done by PRELOCK REPLY, which only sends a reply if the leftmost item in the prelock list is a tuple from where it retrieves the appropriate message identifier. Afterwards, only the client identifier remains, ensuring that replies are only sent once.

The sequence of PRELOCK $\rightarrow$ PRELOCK REPLY $\rightarrow$ PRELOCK etc. is repeated until the PRELOCK rule cannot be applied anymore as there is no identifier in the required locks set that is not in the prelocked processors sets. Note that the acquired set As is always empty while prelock messages are sent.

The client now sends the [Lock] messages to all prelocked processors[6]. Sending is done by LOCK, while the reply is sent by LOCK REPLY. Application of the latter rule appends a new frame at the back of the execution list of the supplier processor. The new frame always starts with the client as the initial owner and the instruction list containing a π symbol to reflect that the following instructions are sent by the client. Both the lock set and the compensation list are empty.

The reply contains a lock on the new frame, which is used by ADD LOCK to add the lock to the set of acquired locks.

The sequence of LOCK $\rightarrow$ LOCK REPLY $\rightarrow$ ADD LOCK $\rightarrow$ LOCK etc. is repeated until the LOCK rule cannot be applied anymore as there is no identifier in the required locks set that is not a supplier in either the acquired or the obtained locks set. Only then can the rule EXEC, listed in fig. 6.17, be applied to invoke the feature body.

6.7.2 Releasing Locks

Releasing locks in D-SCOOP is as simple as sending [Unlock] messages to all the supplier by applying UNLOCK and at some point UNLOCK FINISHED. When processing the [Unlock] message through UNLOCK OK, the supplier is removing the current owner of the indicated frame and adds a υ symbol to the instruction list. When this symbol is processed by one of the POP CLIENT rules, the effects depend on the type of the frame and the compensation list:

1. If the compensation list has multiple entries, then the lock was passed back. The top two items of the compensation list are merged, and the frame is kept.
2. The frame is a result of a lock request. The frame is removed and the compensations are discarded.
3. The frame was created for separate callbacks. The frame is removed and the compensation lists are merged.
4. The frame was created for a local call. The frame is removed and the compensation lists are merged. Also, pending exceptions are propagated to the underlying frame.

[6]While in practice this is done in parallel, we decided to make it sequential so that it can be explained with fewer, simpler rules.

$$\frac{\text{fresh}\,(i)}{\begin{array}{c}\langle c\rangle_{\text{PI}} \vdash \\ \left[\left\langle\left[\textsc{Unlock}\ldots\right]_{\text{IL}}\left\langle\{(s,f)\}\cup A\right\rangle_{\text{AS}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\ldots\right]_{\text{OB}} \\ \rightarrow \\ \left[\left\langle\left[\textsc{Unlock}\ldots\right]_{\text{IL}}\left\langle A\right\rangle_{\text{AS}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\ldots\boxed{\text{Unlock } c,\,f}_i \uparrow s\right]_{\text{OB}}\end{array}}\ \textsc{unlock 1}$$

$$\frac{\begin{array}{c}\text{fresh}\,(i) \\ T \neq \mathfrak{L}\end{array}}{\begin{array}{c}\langle p\rangle_{\text{PI}} \vdash \left[\left\langle\left\langle T\right\rangle_{\text{T}}\left[\textsc{Unlock}\ldots\right]_{\text{IL}}\left\langle\{(s,f)\}\cup I\right\rangle_{\text{IS}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\ldots\right]_{\text{OB}} \\ \rightarrow \\ \left[\left\langle\left\langle T\right\rangle_{\text{T}}\left[\textsc{Unlock}\ldots\right]_{\text{IL}}\left\langle I\right\rangle_{\text{IS}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\ldots\boxed{\text{Unlock } p,\,f}_i \uparrow s\right]_{\text{OB}}\end{array}}\ \textsc{unlock 2}$$

$$\frac{I = \emptyset \vee T = \mathfrak{L}}{\begin{array}{c}\langle p\rangle_{\text{PI}} \vdash \\ \left[\left\langle\left\langle T\right\rangle_{\text{T}}\left[\boxed{\text{Unlock}}\ldots\right]_{\text{IL}}\left\langle\emptyset\right\rangle_{\text{AS}}\left\langle I\right\rangle_{\text{IS}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}} \\ \rightarrow \\ \left[\left\langle\left\langle T\right\rangle_{\text{T}}\left[\ldots\right]_{\text{IL}}\left\langle\emptyset\right\rangle_{\text{AS}}\left\langle I\right\rangle_{\text{IS}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\end{array}}\ \textsc{unlock finished}$$

$$\frac{}{\begin{array}{c}\left[\ldots\left\langle\left\langle f\right\rangle_{\text{FI}}\left[\ldots\right]_{\text{IL}}\left[c\ldots\right]_{\text{OL}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\boxed{\text{Unlock } c,\,f}_m\ldots\right]_{\text{IB}} \\ \rightarrow \\ \left[\ldots\left\langle\left\langle f\right\rangle_{\text{FI}}\left[\ldots v\right]_{\text{IL}}\left[\ldots\right]_{\text{OL}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\ldots\right]_{\text{IB}}\end{array}}\ \textsc{unlock ok}$$

$$\frac{}{\begin{array}{c}\left[\left\langle\left[v\ldots\right]_{\text{IL}}\left[\left(\left[C\right],c\right);\left(\left[D\right],p\right)\ldots\right]_{\text{CL}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}} \\ \rightarrow \left[\left\langle\left[\ldots\right]_{\text{IL}}\left[\left(\left[C;D\right],p\right)\ldots\right]_{\text{CL}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\end{array}}\ \textsc{pop client 1}$$

$$\frac{}{\left[\left\langle\left\langle\mathfrak{S}\right\rangle_{\text{T}}\left[v\ldots\right]_{\text{IL}}\left[\left(\left[C\right],c\right)\right]_{\text{CL}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}} \rightarrow \left[\ldots\right]_{\text{EL}}}\ \textsc{pop client 2}$$

$$\frac{}{\begin{array}{c}\langle p\rangle_{\text{PI}} \vdash \left[\begin{array}{c}\left\langle\left\langle\mathfrak{C}\right\rangle_{\text{T}}\left[v\ldots\right]_{\text{IL}}\left[\left(\left[C\right],c\right)\right]_{\text{CL}}\right\rangle_{\text{F}}; \\ \left\langle\left[\left(\left[D\right],q\right)\ldots\right]_{\text{CL}}\right\rangle_{\text{F}}\ldots\end{array}\right]_{\text{EL}} \\ \rightarrow \\ \left[\left\langle\left[\left(\left[C;D\right],q\right)\ldots\right]_{\text{CL}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\end{array}}\ \textsc{pop client 3}$$

$$\frac{}{\begin{array}{c}\langle p\rangle_{\text{PI}} \vdash \left[\begin{array}{c}\left\langle\left\langle\mathfrak{L}\right\rangle_{\text{T}}\left[v;E\right]_{\text{IL}}\left[\left(\left[C\right],p\right)\right]_{\text{CL}}\right\rangle_{\text{F}}; \\ \left\langle\left[\ldots\right]_{\text{IL}}\left[\left(\left[D\right],q\right)\ldots\right]_{\text{CL}}\right\rangle_{\text{F}}\ldots\end{array}\right]_{\text{EL}} \\ \rightarrow \left[\left\langle\left[E;\ldots\right]_{\text{IL}}\left[\left(\left[C;D\right],q\right)\ldots\right]_{\text{CL}}\right\rangle_{\text{F}}\ldots\right]_{\text{EL}}\end{array}}\ \textsc{pop client 4}$$

Figure 6.10: Inference rules for unlocking

$$\frac{\text{fresh}\,(j)}{\langle c\rangle_{\text{PI}} \vdash \left[\langle\left[\text{PASS}\ \{(l,i)\}\cup L, s\ldots\right]_{\text{IL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\ldots\right]_{\text{OB}} \rightarrow \left[\langle\left[\text{PASS}\ L, s\ldots\right]_{\text{IL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\ldots\boxed{\text{Pass}\ c,i,s}_{j}\uparrow l\right]_{\text{OB}}}\ \text{PASS}$$

$$\frac{}{\left[\langle\left[\text{PASS}\ \emptyset, s\ldots\right]_{\text{IL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}} \rightarrow \left[\langle\left[\ldots\right]_{\text{IL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}}\ \text{PASS FINISHED}$$

$$\frac{}{\left[\langle\left[\ldots\right]_{\text{IL}}\left[c\ldots\right]_{\text{OL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\boxed{\text{Pass}\ c,i,s}\ldots\right]_{\text{IB}} \rightarrow \left[\langle\left[\ldots\pi_s\right]_{\text{IL}}\left[s,c\ldots\right]_{\text{OL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}\left[\ldots\right]_{\text{IB}}}\ \text{PASS OK}$$

$$\frac{}{\left[\langle\left[\pi_c\ldots\right]_{\text{IL}}\left[\ldots\right]_{\text{CL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}} \rightarrow \left[\langle\left[\ldots\right]_{\text{IL}}\left[\left(\left[\epsilon\right], c\right)\ldots\right]_{\text{CL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}}\ \text{PUSH CLIENT}$$

Figure 6.11: Inference rules for lock passing

There is also a RETRACT instruction. This instruction can be used to retract the prelock from a supplier. This is useful is situations where the locking process fails for some reason, and a use case is presented in section 6.10. The RETRACT rules are similar to the LOCK rules explained in section 6.7.1 but without a reply and the creation of a new frame.

6.7.3 Lock Passing

Lock passing is an important part of synchronization. As mentioned in section 3.3.2, lock passing avoids deadlocks in nested synchronous calls. The PASS instruction sends messages (rule PASS) to all suppliers informing them that the lock on the given frame has been passed to another processor. Receiving the message pushes the owner list, which is always used as a stack, with the given processor identifier (rule PASS OK) and adds a pass marker π. The last argument of the synchronous call message $\boxed{\text{SCall}}$ presented in section 6.8.1 is a set of locks the supplier can use. When processed by PUSH CLIENT, the symbol pushes the new lock holder onto the compensation list. Since compensations are applied in reverse, the compensation list is treated like a stack. The π symbol can be seen as a counterpart to the υ symbol: on unlock, υ reduces the owners list again, as stated above in section 6.7.2. This information is used for recovering from failure, which is discussed in section 6.10.

6.7.4 Wait Conditions

It is possible to model wait condition through regular lock → check → unlock cycles until the condition matches. However, this implies a polling-type strategy which is inefficient. What should happen is that the wait condition is only re-evaluated if the state of at least one supplier changes. The rules that specify how wait conditions are evaluated in a pushed fashion are shown in fig. 6.12.

There are three wait states of a processor. The first, $\mathfrak{N}$ for *neutral*, is used if the processor is not waiting for a notification. The second, $\mathfrak{W}_w$ for *waiting*, awaits a

$$\frac{}{[\langle[\Omega\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\textsc{Notify}; \Omega\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}}\ \textsc{next notify}$$

$$\frac{\text{fresh}\,(w)}{[\langle[\Omega\ldots]_{\text{IL}}\langle A\rangle_{\text{As}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\textsc{Await}\ A, w\ldots]_{\text{IL}}\langle A\rangle_{\text{As}}\rangle_{\text{F}}\ldots]_{\text{EL}}}\ \textsc{next await}$$

$$\frac{W \neq \mathfrak{G}}{\begin{array}{c}[\langle\langle f\rangle_{\text{FI}}[\textsc{Await}\ R, w\ldots]_{\text{IL}}\langle\{(s,f)\}\cup A\rangle_{\text{As}}\rangle_{\text{F}}\ldots]_{\text{EL}}\\ \langle W\rangle_{\text{Ws}}[\ldots]_{\text{OB}}\\ \to\\ [\langle\langle f\rangle_{\text{FI}}[\textsc{Await}\ R\ldots]_{\text{IL}}\langle A\rangle_{\text{As}}\rangle_{\text{F}}\ldots]_{\text{EL}}\\ \langle\mathfrak{W}_w\rangle_{\text{Ws}}[\ldots\boxed{\textsc{Await}\ c, f}_w \uparrow s]_{\text{OB}}\end{array}}\ \textsc{await 1}$$

$$\frac{}{\begin{array}{c}\langle\mathfrak{G}\rangle_{\text{Ws}} \vdash\\ [\langle\langle f\rangle_{\text{FI}}[\textsc{Await}\ R, w\ldots]_{\text{IL}}\langle\{(s,f)\}\cup A\rangle_{\text{As}}\rangle_{\text{F}}\ldots]_{\text{EL}}[\ldots]_{\text{OB}} \to\\ [\langle\langle f\rangle_{\text{FI}}[\textsc{Await}\ R\ldots]_{\text{IL}}\langle A\rangle_{\text{As}}\rangle_{\text{F}}\ldots]_{\text{EL}}[\ldots\boxed{\textsc{Unlock}\ c, f}_w \uparrow s]_{\text{OB}}\end{array}}\ \textsc{await 2}$$

$$\frac{}{\begin{array}{c}[\langle[\textsc{Await}\ R, w\ldots]_{\text{IL}}\langle\emptyset\rangle_{\text{As}}\rangle_{\text{F}}\ldots\langle\mathfrak{R}\rangle_{\text{Ws}}]_{\text{EL}} \to\\ [\langle[\textsc{Exec}\ R, \emptyset, \emptyset\ldots]_{\text{IL}}\langle\emptyset\rangle_{\text{As}}\rangle_{\text{F}}\ldots\langle\mathfrak{N}\rangle_{\text{Ws}}]_{\text{EL}}\end{array}}\ \textsc{retry}$$

$$\frac{o \in \mathbb{N}}{\begin{array}{c}[\ldots\langle\langle f\rangle_{\text{FI}}[\ldots]_{\text{IL}}[c]_{\text{OL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\langle N\rangle_{\text{Ns}}[\boxed{\textsc{Await}\ (c, f)}_w\ldots]_{\text{IB}}\\ \to [\ldots\langle\langle f\rangle_{\text{FI}}[\ldots v]_{\text{IL}}[\epsilon]_{\text{OL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\langle N\cup\{(c,w)\}\rangle_{\text{Ns}}[\ldots]_{\text{IB}}\end{array}}\ \textsc{await ok}$$

$$\frac{}{\begin{array}{c}[\langle[\textsc{Notify}\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\langle N\cup\{(c,w)\}\rangle_{\text{Ns}}[\ldots]_{\text{OB}}\\ \to\\ [\langle[\textsc{Notify}\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\langle N\rangle_{\text{Ns}}[\ldots\boxed{\textsc{Ready}}_w \uparrow c]_{\text{OB}}\end{array}}\ \textsc{notify}$$

$$\frac{}{\begin{array}{c}\langle\emptyset\rangle_{\text{Ns}} \vdash\\ [\langle[\textsc{Notify}\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\end{array}}\ \textsc{notify fin.}$$

$$\frac{}{[\boxed{\textsc{Ready}}_w; \ldots]_{\text{IB}}\langle\mathfrak{W}_w\rangle_{\text{Ws}} \to [\ldots]_{\text{IB}}\langle\mathfrak{R}\rangle_{\text{Ws}}}\ \textsc{ready}$$

$$\frac{W \neq \mathfrak{W}_w}{\langle W\rangle_{\text{Ws}} \vdash [\boxed{\textsc{Ready}\ w}_w; \ldots]_{\text{IB}} \to [\ldots]_{\text{IB}}}\ \textsc{ignore}$$

Figure 6.12: Inference rules for wait conditions

notification with identifier w. The third, $\mathfrak{R}$ for *ready,* signals a received notification and allows the processor to resume, which will put it back to $\mathfrak{N}$.

Due to routine abstraction (see section 6.8.1), the state change is not observable, but the notification is. Therefore, Ω can produce the NOTIFY instruction which notifies waiting processors by sending [Ready] messages according to NOTIFY and NOTIFY FIN.. These messages are sent to clients that registered for notification by sending an [Await] message instead of an [Unlock] message. This causes the client to be added to the notification set in the Ns cell. This is done by AWAIT rules processing the AWAIT instruction produced by Ω through NEXT AWAIT. These two rules are very similar to the UNLOCK and NEXT UNLOCK rules. In the client, the W cell tracks whether the processor is currently waiting; the cell is reset when receiving a [Ready] message and applying READY. The AWAIT 2 rule is used if the processor was already notified but did not yet send all the [Await] messages. In this case, the notified processor sends regular [Unlock] messages to the remaining processors instead of further [Await] messages. If a [Ready] message arrives after the processor was already notified, the message is simply ignored through rule IGNORE.

If a supplier receives a [Prelock] message from a client that is in the notification set, this set entry is removed.

6.8 Execution

6.8.1 Feature abstraction

As mentioned before, the semantics of the actual feature execution is abstract. We assume that a feature can take the following actions:

- make an internal action
- make a non-separate call
- make a separate synchronous call
- make a separate asynchronous call
- spawn a new processor
- register a compensation call
- throw an exception
- unlock and finish

The internal action subsumes all decision-making within the feature as well as all changes to the internal, not represented state. This has no influence on the visible state, which is why there is no inference rule for internal actions. However, this abstraction also removes the possibility to infer, from the current state, what action a feature is taking next, making this decision nondeterministic. To model this nondeterminism, the symbol Ω abstractly represents the feature: it can be seen as a yet unknown list of actions. This symbol produces non-deterministically the different actions a feature can take as listed above.

The possible actions are represented as inference rules in fig. 6.13. By application of NEXT SPAWN, a feature can create a new processor. The rule NEXT UNLOCK represents the end of a feature execution by replacing the Ω symbol with an instruction to unlock the supplier frames. Compensation and exceptions are discussed in section 6.10.

$$\frac{s \in \text{suppliers}(A \cup I) \quad R \subseteq \mathbb{P}}{\langle \mathbb{P} \rangle_{\text{Ps}} \vdash [\langle [\Omega \ldots]_{\text{IL}} \langle A \rangle_{\text{As}} \langle I \rangle_{\text{Is}} \rangle_{\text{F}} \cdots]_{\text{EL}} \rightarrow [\langle [\textsc{ACall}\ s, R; \Omega \ldots]_{\text{IL}} \langle A \rangle_{\text{As}} \langle I \rangle_{\text{Is}} \rangle_{\text{F}} \cdots]_{\text{EL}}} \text{ NEXT ACALL}$$

$$\frac{R \subseteq \mathbb{P}}{\langle \mathbb{P} \rangle_{\text{Ps}} \langle c \rangle_{\text{PI}} \vdash [\langle [\Omega \ldots]_{\text{IL}} \rangle_{\text{F}} \cdots]_{\text{EL}} \rightarrow [\langle [\textsc{SCall}\ c, R; \Omega \ldots]_{\text{IL}} \rangle_{\text{F}} \cdots]_{\text{EL}}} \text{ NEXT SCALL 1}$$

$$\frac{(s,i) \in A \cup I \quad S = (A \cup I) \setminus \{(s,i)\} \quad R \subseteq P}{\langle P \rangle_{\text{Ps}} \vdash [\langle [\Omega \ldots]_{\text{IL}} \langle A \rangle_{\text{As}} \langle I \rangle_{\text{Is}} \rangle_{\text{F}} \cdots]_{\text{EL}} \rightarrow [\langle [\textsc{Pass}\ S, s; \textsc{SCall}\ s, R; \Omega \ldots]_{\text{IL}} \langle A \rangle_{\text{As}} \langle I \rangle_{\text{Is}} \rangle_{\text{F}} \cdots]_{\text{EL}}} \text{ NEXT SCALL 2}$$

$$\frac{}{[\langle [\Omega \ldots]_{\text{IL}} \rangle_{\text{F}} \cdots]_{\text{EL}} \rightarrow [\langle [\textsc{Spawn}; \Omega \ldots]_{\text{IL}} \rangle_{\text{F}} \cdots]_{\text{EL}}} \text{ NEXT SPAWN}$$

$$\frac{}{[\langle [\Omega \ldots]_{\text{IL}} \rangle_{\text{F}} \cdots]_{\text{EL}} \rightarrow [\langle [\textsc{Unlock} \ldots]_{\text{IL}} \rangle_{\text{F}} \cdots]_{\text{EL}}} \text{ NEXT UNLOCK}$$

Figure 6.13: Inference rules for routine abstraction

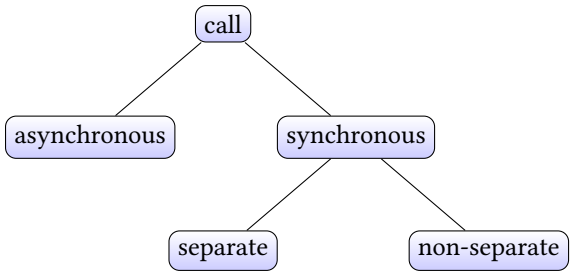


Figure 6.14: Call categories

Generating calls

Calls can be divided into two general categories: synchronous and asynchronous calls. Synchronous calls have the property that the locks are passed along to the target processor. Client and supplier processor of a synchronous call can be the same, in which case we use the term non-separate call. Section 6.8.1 illustrates this categorization. Handling of calls differs based on whether the call is (separate) asynchronous, separate synchronous or non-separate synchronous.

A feature can make calls, which is represented by the application of one of the rules NEXT SCALL 1, NEXT SCALL 2 or NEXT ACALL. In all three cases, the arguments are a subset of the processors in the system. But there are also some subtle differences for these calls. An asynchronous call uses a different instruction symbol, ACALL, to indicate that the call is not blocking. The supplier cannot be the current processor, of which there is never a lock in the locks set, since an asynchronous call is always separate. There is also a distinction between non-separate and synchronous calls: separate lock-passing, through the instruction PASS, happens only in the case of separate synchronous calls and not for non-separate synchronous calls. Note that the lock on the supplier is not sent with lock passing, but a new lock on the client is.

This is only the generation of the calls. Handling of the generated call is discussed in the next section.

Locked arguments and lock passing. Note that the SCOOP model and the concrete D-SCOOP system described in chapter 3 and chapter 5 specify that lock passing, and therefore a synchronous call, happens whenever an argument of a call to a separate target is locked. This is a programming language decision that is not strictly required: in fact, in [49], the decision to pass locks is based on whether the argument is controlled, not whether it is locked. In section 5.10, we propose a simple language change to force an asynchronous call. So semantically, an asynchronous call may involve arguments, that is, needed locks, that are held by the caller. Conversely, the semantics allows that a separate command call can have no required locks and still be synchronous, which represents command calls to passive regions or possible future language extensions to force a synchronous call or lock passing.

6.8.2 Making calls

The different types of calls need to be handled i different ways. This is achieved by the rules shown in fig. 6.15.

Non-separate calls. Non-separate calls are always synchronous, which is why they are represented by the SCALL instruction. According to the rule SCALL 1 shown in fig. 6.15, non-separate calls are executed by creating a new frame with the corresponding EXEC instruction. Non-separate lock passing is silent: the suppliers are not notified when it happens and also do not get an [Unlock] message. Also, the locks are immediately put into the inherited set, so the second argument of the EXEC instruction is empty.

Asynchronous calls. Asynchronous calls are always separate, that is, the supplier is different from the client. Such a call is, according to the rule ACALL, handled by sending a [ACall] message to the supplier containing the client, the identifier of the frame in the lock on the supplier and the arguments. No changes are made to the set

$$\frac{\text{fresh}\,(i)}{\begin{array}{c}\langle c\rangle_{\text{PI}} \vdash \left[\left\langle\left[\text{SCALL}\ c, R{\ldots}\right]_{\text{IL}}\langle I\rangle_{\text{IS}}\langle A\rangle_{\text{AS}}\right\rangle_{\text{F}}{\ldots}\right]_{\text{EL}} \\ \rightarrow \\ \left[\left\langle \begin{array}{c}\left[\text{EXEC}\ R \setminus (I \cup A \cup \{c\}), \emptyset, \emptyset; v\right]_{\text{IL}} \\ [i]_{\text{FI}}\,[\epsilon]_{\text{OL}}\left[([\epsilon]\,, c)\right]_{\text{CL}}\langle I \cup A\rangle_{\text{IS}}\langle\emptyset\rangle_{\text{AS}}\end{array}\right\rangle_{\text{F}} \right]_{\text{EL}} \\ ;\left\langle[{\ldots}]_{\text{IL}}\langle I\rangle_{\text{IS}}\langle A\rangle_{\text{AS}}\right\rangle_{\text{F}}\end{array}}\ \text{SCALL 1}$$

$$\frac{\begin{array}{c}(s,i) \in I \cup A \\ \text{fresh}\,(m)\end{array}}{\begin{array}{c}\langle c\rangle_{\text{PI}} \vdash \left[\left\langle\left[\text{ACALL}\ s, R{\ldots}\right]_{\text{IL}}\langle I\rangle_{\text{IS}}\langle A\rangle_{\text{AS}}\right\rangle_{\text{F}}{\ldots}\right]_{\text{EL}}\left[{\ldots}\right]_{\text{OB}} \\ \rightarrow \\ \left[\left\langle\left[m \downarrow s{\ldots}\right]_{\text{IL}}\langle I\rangle_{\text{IS}}\langle A\rangle_{\text{AS}}\right\rangle_{\text{F}}{\ldots}\right]_{\text{EL}}\left[{\ldots}\boxed{\text{ACall}\ c, i, R}_{m} \uparrow s\right]_{\text{OB}}\end{array}}\ \text{ACALL}$$

$$\frac{\begin{array}{c}\text{fresh}\,(i,j) \\ (s,f) \in I \cup A \\ P = (I \cup A \cup \{(c,i)\}) \setminus \{(s,f)\}\end{array}}{\begin{array}{c}\langle c\rangle_{\text{PI}} \vdash \left[\left\langle\left[\text{SCALL}\ s, R{\ldots}\right]_{\text{IL}}\langle I\rangle_{\text{IS}}\langle A\rangle_{\text{AS}}\right\rangle_{\text{F}}{\ldots}\right]_{\text{EL}}\left[{\ldots}\right]_{\text{OB}} \\ \rightarrow \\ \left[\left\langle[i]_{\text{FI}}\,[\epsilon]_{\text{IL}}\,[s]_{\text{OL}}\left[\left[([\epsilon]\,, s)\right]_{\text{CL}}\right]_{\text{CL}}\langle\emptyset\rangle_{\text{IS}}\langle\emptyset\rangle_{\text{AS}}\right\rangle_{\text{F}}\right. \\ \left.;\left\langle[j \downarrow s{\ldots}]_{\text{IL}}\langle I\rangle_{\text{IS}}\langle A\rangle_{\text{AS}}\right\rangle_{\text{F}}{\ldots}\right]_{\text{EL}} \\ \left[{\ldots}\boxed{\text{SCall}\ c, f, R, P}_{j} \uparrow s\right]_{\text{OB}}\end{array}}\ \text{SCALL 2}$$

Figure 6.15: Inference rules for calls

$$\frac{R = N \setminus (\text{supplier}\,(O) \cup \{s\})}{\begin{array}{c}\langle s\rangle_{\text{PI}} \vdash \\ \left[\ldots\langle\langle i\rangle_{\text{FI}}\,[\ldots]_{\text{IL}}\,[c\ldots]_{\text{OL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}\,\left[\boxed{\text{SCall}\ c, i, N, O}_j\ldots\right]_{\text{IB}} \\ \rightarrow \\ \left[\ldots\langle\langle i\rangle_{\text{FI}}\,[\ldots\textsc{Exec}\ R, O, \emptyset; \textsc{Return}\ c, j]_{\text{IL}}\,[c\ldots]_{\text{OL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}} \\ [\ldots]_{\text{IB}}\end{array}}\ \textsc{enq. scall}$$

$$\frac{}{\begin{array}{c}\langle s\rangle_{\text{PI}} \vdash \\ \left[\ldots\langle\langle i\rangle_{\text{FI}}\,[\ldots]_{\text{IL}}\,[c\ldots]_{\text{OL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}} \\ \left[\boxed{\text{ACall}\ c, i, N}_j\ldots\right]_{\text{IB}}\,[\ldots]_{\text{OB}} \\ \rightarrow \\ \left[\ldots\langle\langle i\rangle_{\text{FI}}\,[\ldots\textsc{Exec}\ N \setminus \{s\}, \emptyset, \emptyset]_{\text{IL}}\,[c\ldots]_{\text{OL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}} \\ [\ldots]_{\text{IB}}\,[\ldots\boxed{\text{OK}}_j \uparrow c]_{\text{OB}}\end{array}}\ \textsc{enq. acall}$$

$$\frac{}{\begin{array}{c}\left[\langle\left[\textsc{Return}\ c, i\ldots\right]_{\text{IL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}\,[\ldots]_{\text{OB}} \\ \rightarrow \\ \left[\langle\left[\ldots\right]_{\text{IL}}\rangle_{\text{F}}\ldots\right]_{\text{EL}}\,[\ldots\boxed{\text{OK}}_i \uparrow c]_{\text{OB}}\end{array}}\ \textsc{return}\ 1$$

Figure 6.16: Inference rules for enqueuing of calls

of locks, the supplier needs to acquire all necessary locks for the call. The rule sends the message and blocks the client until it gets the OK from the supplier, which is sent when the call has been successfully enqueued, but not necessarily when it has been executed. This has a practical reason: if the supplier is unreachable because of failure, the client gets notified immediately. If the message was sent and the processor does not wait for the response, a failed supplier is only detected with the next synchronous call, if there is one.

Synchronous separate calls. Synchronous separate calls are more complex than other types of calls and are handled by rule SCALL 2. Note that before this rule is applied, the PASS instruction, detailed in section 6.7.3 has been handled. On a synchronous call, the client blocks and sends along all its locks to the supplier. For supporting direct or indirect callbacks, the client adds a new execution frame in *front* of the execution list, and passes the lock on it to the supplier alongside the other locks. The new frame is the same as the one in LOCK REPLY mentioned in section 6.7.1, so it has an empty lock set, which ensures that asynchronous callbacks do not have access to the set of locks. As in the case of asynchronous calls, the SCall message is sent and the client blocks until it gets a reply. However, unlike asynchronous calls, the reply is sent when the feature has finished execution.

6.8.3 Enqueuing calls

While non-separate calls are executed in place (see section 6.8.1), incoming requests for asynchronous and synchronous execution are enqueued in the corresponding in-

$$\frac{R \subseteq \text{suppliers}\,(A)}{\left[\langle\left[\textsc{Exec}\ R, O, R{\ldots}\right]_{\text{IL}}\langle I\rangle_{\text{Is}}\langle A\rangle_{\text{As}}\rangle_{\text{F}}\cdots\right]_{\text{EL}} \xrightarrow{} \left[\langle\left[\Omega{\ldots}\right]_{\text{IL}}\langle I \cup O\rangle_{\text{Is}}\langle A\rangle_{\text{As}}\rangle_{\text{F}}\cdots\right]_{\text{EL}}}\ \textsc{exec}$$

Figure 6.17: Inference rule for execution

struction queue of the supplier. In D-Scoop, the message queue does not also serve as the request queue, as opposed to Queue-of-Queues.

The rules for enqueuing call requests, ENQ. SCALL and ENQ. ACALL in fig. 6.16, are very similar: they both enqueue an Exec instruction in the instruction queue. The first argument of the instruction is the set of processors of the arguments of the call, that is, the set of needed locks; the second argument is the set of passed locks. The latter is empty in the asynchronous case and has at least one element in the synchronous case. Note that the request is only enqueued if the owner matches. This is required in the case that a processor disappears, so that the compensation code is executed before more requests are enqueued. This can happen in the following case:

Processor A acquires a lock on processor C. It then passes it to processor B. Processor B disappears. Processor A notices this and assumes that it can again use the lock. It then issues a call to processor C. However, processor C did not yet notice that processor B disappeared and therefore did not clean up by running compensations and adding a υ, as we can see in section 6.10. If it enqueues the call from A, the call would end up being treated as if it was from B.

The reader might argue that keeping the message in the in-box may cause a deadlock, but this is not the case since the rule for missing clients immediately removes the owner and does not involve the in-box. So when the message is stuck in the in-box, the owner can be removed which then allows the message to be handled.

Asynchronous call requests are replied to immediately, whereas synchronous call requests append a Return instruction for sending back the result via RETURN 1 later. Since objects have no representation in D-Scoop semantics, the FIXME message has no arguments, it just unblocks the waiting client.

6.8.4 Feature application

Feature application is handled by the EXEC rule in fig. 6.17, which removes the instruction with the same name and replaces it with Ω. The symbol represents the observable actions of the feature explained in section 6.8.1.

6.9 Exceptions

Routine abstraction, as explained in section 6.8.1, also encompasses the execution of a rescue or catch clause. However, the semantics needs to define how exceptions are propagated, especially across the network. Exceptions and asynchronous execution have been investigated in [45]. We apply the results of this paper in our semantics for exceptions with respect to the underlying message-passing mechanism.

In addition to regular exceptions raised by the software or the operating system, D-Scoop also introduces exception due to broken connections. For example, if the

$$\frac{}{[\langle[\xi;\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}}\ \xi\ \xi$$

$$\frac{}{[\langle[\xi;\pi_c\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\pi_c;\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}}\ \xi\ \pi$$

$$\frac{}{[\langle[\xi;\upsilon\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\upsilon;\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}}\ \xi\ \upsilon$$

$$\frac{}{\begin{array}{c}[\langle[\xi;\textsc{Retract}\ L\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to \\ [\langle[\textsc{Retract}\ L;\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\end{array}}\ \xi\ \textsc{retract}$$

$$\frac{}{\begin{array}{c}[\langle[\xi;\textsc{Unlock}\ L\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to \\ [\langle[\textsc{Unlock}\ L;\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\end{array}}\ \xi\ \textsc{unlock}$$

$$\frac{}{[\langle[\xi;\textsc{Revert}\ c\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\textsc{Revert}\ c;\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}}\ \xi\ \textsc{revert}$$

$$\frac{}{[\langle[\Omega\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}}\ \textsc{next}\ \xi$$

$$\frac{}{[\langle[\xi;\Omega\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\langle[\xi;\Omega\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}}\ \textsc{catch}$$

$$\frac{}{\begin{array}{c}[\langle[\xi;\textsc{Exec}\ R,O,A,P\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to \\ [\langle[\textsc{Retract}\ P;\textsc{Unlock}\ O;\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\end{array}}\ \textsc{cancel call}$$

$$\frac{}{\begin{array}{c}[\langle[\xi;\textsc{Return}\ c,i\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\ [\ldots]_{\text{OB}} \to \\ [\langle[\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\ [\ldots\boxed{\text{FAIL}}_i \uparrow c]_{\text{OB}}\end{array}}\ \xi\ \textsc{return}$$

$$\frac{}{[\langle[\xi]_{\text{IL}}\ [\epsilon]_{\text{OL}}\rangle_{\text{F}}\ldots]_{\text{EL}} \to [\ldots]_{\text{EL}}}\ \textsc{next queue 2}$$

$$\frac{}{\begin{array}{c}[\ldots\langle[\ldots i \downarrow s\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\ [\boxed{\text{FAIL}}_i\ldots]_{\text{IB}} \\ \to [\ldots\langle[\ldots\xi\ldots]_{\text{IL}}\rangle_{\text{F}}\ldots]_{\text{EL}}\ [\ldots]_{\text{IB}}\end{array}}\ \textsc{recv. fail}$$

Figure 6.18: Inference rules for exceptions

target processor of a call is no longer available, an exception is raised. But also if a lock on a processor cannot be acquired due to connection problems, the equivalent of a precondition violation exception is raised, which also precludes the execution of the body. Although there are similarities to a failed contract, exceptions due to technical problems are not considered as such since they can occur in a correct system, whereas a contract violation always indicates an error in specification or implementation. The following paragraphs gives an overview of those situations.

Asynchronous calls. An exception occurs if the target processor disappears before the message was processed. Absence of this exception, however, is not a guarantee that the call has been finished or even started. Exceptions due to application of the call either due to problems arising during locking or due to internal behavior are stored within the frame on the supplier. If the supplier frame contains an exception, the processor is dirty and all further asynchronous calls are dropped silently.

Synchronous calls. In addition to the cases described for asynchronous calls, exceptions arising due to the application of synchronous calls are propagated to the client, as are exceptions stored in the supplier frame from previous asynchronous calls. In this case, the synchronous call is not applied, instead, the stored exceptions are propagated to the client while removing them from the supplier. Therefore, after a synchronous call, a processor is no longer dirty. Note that in the case of lock passing, the stored exception is not removed and will be propagated to the client of the client if and when it issues a synchronous call.

Syntax. We use the symbol ξ to denote exceptions, and they appear only in the instruction list and as an argument to a [FAIL] message.

Semantics. The first few rules in fig. 6.18 are simple interactions with other symbols in the instruction list. If two exceptions end up next to each other, for example due to multiple failures in the system or a failure during revert, they are merged into one. Locking and unlocking, as well as their supplier counterparts, the π and υ symbols, are not affected by exceptions.

Exceptions can be raised by the program itself, which is represented by NEXT ξ. A running routine may CATCH an exception and remove it from the instruction list. An application of CATCH immediately followed by NEXT ξ indicates that the routine simply forwarded the exception without taking any observable measures.

As mentioned above, the rule CANCEL CALL removes a pending execution from the instruction list. Already acquired prelocks are RETRACTed and obtained locks are unlocked. Note that this rule will not apply for non-separate calls, as it is not possible for an Ω to produce anything if it is behind a ξ. Also, the RETURN instruction of a synchronous call is not dropped, as it is added after the EXEC instruction if a synchronous call request is processed (see fig. 6.16).

ξ RETURN sends a [FAIL] message. An interpretation of this message is that it contains the exception, which is removed from the instruction list. Since this is a synchronous call, the client cannot have added more calls. Due to the rule RECV. FAIL, the exception is propagated to the client, which is therefore aware of the error and choses its further calls accordingly.

The NEXT QUEUE 2 rule allows for a frame to be removed if it contains an exception but does not have a client anymore. The exception is silently dropped since the client

$$\frac{}{\langle \mathbb{P} \cup \{p\} \rangle_{\text{Ps}} \langle \langle p \rangle_{\text{Pi}} \rangle_{\text{P}} \to \langle \mathbb{P} \rangle_{\text{Ps}}} \textsc{disappear}$$

$$\frac{s \notin \mathbb{P}}{\langle \mathbb{P} \rangle_{\text{Ps}} \vdash [M \uparrow s \ldots]_{\text{Ob}} \to [\ldots]_{\text{Ob}}} \textsc{send error}$$

$$\frac{s \notin \mathbb{P}}{\langle \mathbb{P} \rangle_{\text{Ps}} \vdash [\ldots \langle [\ldots i \downarrow s \ldots]_{\text{Il}} \rangle_{\text{F}} \ldots]_{\text{El}} \to [\ldots \langle [\ldots \xi \ldots]_{\text{Il}} \rangle_{\text{F}} \ldots]_{\text{El}}} \textsc{recv. error}$$

$$\frac{\begin{array}{c} c \notin \mathbb{P} \\ I = \epsilon \vee I = \xi \end{array}}{\begin{array}{c} \langle \mathbb{P} \rangle_{\text{Ps}} [\epsilon]_{\text{Ib}} \vdash [\langle [I]_{\text{Il}} [([C], c) \ldots]_{\text{Cl}} [c \ldots]_{\text{Ol}} \rangle_{\text{F}} \ldots]_{\text{El}} \\ \to \\ [\langle [\textsc{Revert}; \xi]_{\text{Il}} [([C], c) \ldots]_{\text{Cl}} [\ldots]_{\text{Ol}} \rangle_{\text{F}} \ldots]_{\text{El}} \end{array}} \textsc{client missing}$$

$$\frac{c \notin \mathbb{P}}{\langle \mathbb{P} \rangle_{\text{Ps}} \vdash [c \ldots]_{\text{Pl}} \to [\ldots]_{\text{Pl}}} \textsc{prelock abort}$$

$$\frac{}{\begin{array}{c} \langle c \rangle_{\text{Pi}} \vdash [\langle [\textsc{Retract } \{p\} \cup P \ldots]_{\text{Il}} \rangle_{\text{F}} \ldots]_{\text{El}} \to \\ [\langle [\boxed{\text{Retract } c} \uparrow p; \textsc{Retract } P \ldots]_{\text{Il}} \rangle_{\text{F}} \ldots]_{\text{El}} \end{array}} \textsc{retract}$$

$$\frac{}{\begin{array}{c} [\langle [\textsc{Retract } \emptyset \ldots]_{\text{Il}} \rangle_{\text{F}} \ldots]_{\text{El}} \to \\ [\langle [\ldots]_{\text{Il}} \rangle_{\text{F}} \ldots]_{\text{El}} \end{array}} \textsc{retract fin.}$$

$$\frac{}{[\boxed{\text{Retract } c}_m \ldots]_{\text{Ib}} [c \ldots]_{\text{Pl}} \to [\ldots]_{\text{Ib}} [\ldots]_{\text{Pl}}} \textsc{retract ok}$$

Figure 6.19: Inference rules for disappearance.

of the next frame does not have the context to deal with it. This is in line with [45].

6.10 Resilience

Until now, the semantics of D-Scoop is based on the assumption that processors are always available. This is not the case in many distributed settings, so the semantics needs to deal with failures. This section extends the core semantics with rules handling the scenario when a node disappears. In this case, both client and supplier need to be informed. For the client, this can be done through exceptions, while the supplier needs a more advanced mechanism[7].

[7] Compensation could serve as an alternative to classic exception handling — which is essentially a form of `goto` instruction — as proposed in section 5.4.

6.10.1 Disappearance

Nodes can disappear at any time, which is described by the rule DISAPPEAR in fig. 6.19. The rule removes the entire processor cell from the configuration and the processor identifier from the set of all processor identifiers. The latter step is needed so that rules can check for absence of processors as it is impossible to check for the absence of a cell, as explained in section 6.3.1.

Since all interactions between processors except spawning is through message passing, there are two main scenarios where a missing processor is relevant: when sending a message to the missing processor and when waiting for a reply from the same. But there are also more subtle cases: First, if a client of a processor disappears without unlocking its frame, the supplier would be stuck forever. And second, if a processor disappears during prelock, other processors might be barred from acquiring locks. The D-Scoop semantics handles these cases.

Disappearance of a processor does not need to be a failure scenario. In many distributed systems, it is perfectly normal for nodes to shut down. Also, some processors, once they fulfilled their purpose, are no longer needed. However, a processor that does an orderly shut down should never have locks on, or frames owned by, other processors.

Semantics

If the recipient of a message in the out-box of a processor is no longer available, the message is simply dropped because of SEND ERROR. The reason for this behavior is simple: if the request was asynchronous then the context of this request is already lost, but if it was synchronous, the error is handled by RECV. ERROR the same as if the processor disappeared after receiving the request but before replying. The rule CLIENT MISSING starts the compensation mechanism (discussed in the next section) if a client disappeared. Finally, PRELOCK ABORT simply removes prelocks of missing processors.

6.10.2 Compensation

Errors in a supplier are handled as exceptions in the client, which is the mechanism in Scoop as well as many other common object-oriented languages. However, with distributed systems it is possible that a client disappears. Instead of simply cleaning up and continuing with the next execution frame, D-Scoop uses a more advanced technique: compensation.

The client has the complete context in which it requires services from the supplier. It waits for the call to succeed or fail, and is prepared for the latter. In the case of failure, it can mitigate the problem. The role of the supplier is to receive and execute calls, it is not aware of the context and purpose of the calls. The only information the supplier has is whether the client is still interested in sending more requests: whether the client has unlocked the frame or not.

Transactions

For D-Scoop, we interpret the frame as a means to ensure non-interference, as in Scoop. While systems such as Software Transactional Memory [27] use transactions for optimistic locking, the Scoop locking scheme is pessimistic. We therefore also use transactions only for mitigating failure and not for concurrency control. A D-Scoop

transaction starts with the insertion of a new frame and ends when the frame is removed. This is an over-simplification since effects, and therefore the compensations, of callbacks are merged back when the frame is removed. The occurrence of π symbols and υ mark sub-transactions that can be reverted selectively to only undo the changes from one client, not from the others that owned the frame before. Exceptions do not cause a transaction to revert.

This interpretation is based on the assumption that a client has a goal that it tries to achieve with a series of calls. If a call is missing due to a disappearing client, the supplier is left in an unwanted intermediate state and should recover. If the missing client was itself a supplier and received the lock with lock passing, the changes that were caused before the lock was passed should be preserved, that is, not compensated.

Compensations

If the client disappears before it unlocks, the supplier can take corrective measures appropriate for the calls that it already executed within the frame. Usually, these measures are supposed to revert the changes stemming from the execution of the calls, hence the name *compensations*. It differs from rescue clauses in that they are executed outside of the feature context. At this point, the supplier has no locks, so the compensations take the form of synchronous non-separate calls.

Upon detecting that a client disappears, the supplier executes the registered compensations. Note that when compensations are executed, the lock set is empty, which is why the REGISTER instructions have no target: the call is always non-separate but may include separate arguments. As discussed in section 6.8.1, Ω produces register instructions nondeterministically. Execution of the registered compensations is done in reverse order of their registration. This is required since a compensation that is registered later might require a state that is reverted by a compensation registered earlier.

While it is possible that the programming language allows or even requires the programmer to manually write the code for the compensations, it is also possible to generate compensations automatically to revert all changes in memory. This can be a variation of Software Transactional Memory [27], where there is only a write-set and no read-set, or by using a copy-on-write memory model, which is intuitive for functional paradigms. We chose not to assume automatic reversal since rollback of already visible changes cannot be automated as it is very application-specific, and this approach subsumes automatic and manual compensation.

Data Structures

Compensations are added to a list of tuples called the *compensation list*, which is part of the frame. The tuple contains the list of compensation for the current transactional frame, as well as the identifier of the client. Both lists are accessed as a stack, that is, items are appended and removed from the left. The usage of two stacks is necessary in the presence of lock passing, where it makes sense to do partial rollbacks: only the top part of the stack of compensations is used for the rollback, so that a possible client of the failed client does not have its effects on the supplier reverted. Every time the lock on the queue is passed, a new stack is put on top. If a queue is unlocked, the two topmost stacks are merged: the actions of the supplier are merged with the actions of the client.

$$\frac{A \subseteq P}{\begin{array}{c}[P]_{\mathrm{P}} \vdash \\ [\langle [\Omega \ldots]_{\mathrm{IL}} \rangle_{\mathrm{F}} \ldots]_{\mathrm{EL}} \\ \rightarrow \\ [\langle [\textsc{Register}\ A; \Omega \ldots]_{\mathrm{IL}} \rangle_{\mathrm{F}} \ldots]_{\mathrm{EL}}\end{array}}\ \textsc{next register}$$

$$\frac{}{\begin{array}{c}[\langle [\textsc{Register}\ A \ldots]_{\mathrm{IL}}\ [([C], c) \ldots]_{\mathrm{CL}} \rangle_{\mathrm{F}} \ldots]_{\mathrm{EL}} \\ \rightarrow \\ [\langle [\ldots]_{\mathrm{IL}}\ [([A; C], c) \ldots]_{\mathrm{CL}} \rangle_{\mathrm{F}} \ldots]_{\mathrm{EL}}\end{array}}\ \textsc{register}$$

$$\frac{}{\begin{array}{c}\langle s \rangle_{\mathrm{PS}} \vdash \\ [\langle [\textsc{Revert} \ldots]_{\mathrm{IL}}\ [([\{A\}; C], c) \ldots]_{\mathrm{CL}} \rangle_{\mathrm{F}} \ldots]_{\mathrm{EL}} \\ \rightarrow \\ [\langle [\textsc{SCall}\ s, \{A\}; \textsc{Revert} \ldots]_{\mathrm{IL}}\ [([C], c) \ldots]_{\mathrm{CL}} \rangle_{\mathrm{F}} \ldots]_{\mathrm{EL}}\end{array}}\ \textsc{revert}$$

$$\frac{}{\begin{array}{c}[\langle [\textsc{Revert} \ldots]_{\mathrm{IL}}\ [([\epsilon], c) \ldots]_{\mathrm{CL}} \rangle_{\mathrm{F}} \ldots]_{\mathrm{EL}} \\ \rightarrow \\ [\langle [v \ldots]_{\mathrm{IL}}\ [([\epsilon], c) \ldots]_{\mathrm{CL}} \rangle_{\mathrm{F}} \ldots]_{\mathrm{EL}}\end{array}}\ \textsc{revert finished}$$

Figure 6.20: Inference rules for compensation

For example, we assume a very simple system. We have a remote processor called A that holds a cell containing an integer. There is a processor B that is currently a client of A, that is, has a lock on it. It sets the cell to some value. Afterwards, it makes a synchronous call to yet another processor C. During the execution of C, which received the lock on A from B, processor C queries A for the value of the cell and then puts the incremented value into B. Processor C then fails before unlocking A. Processor A now undoes the increment action by C. Processor B receives an exception because of the failure of C. It can now assume that the state of A is that immediately before the call to C, so the cell still contains the value set by B.

If all changes would have been rolled back, B would have to re-do those. These changes might be far removed from the feature that made the call to C and has the rescue clause that mitigates the failure from the client side. It might even be possible that the changes were caused by a client of B and B received the lock on A through lock passing. In this case, B has no way of knowing that the compensation caused by C puts A into a state from before its own execution, this state might also violate preconditions that held before. Rolling back all the way to the beginning of the transaction would require all failure to cascade back to the root.

Semantics

If a routine produces a Register instruction through the application of next register, its arguments are added as a set to the top compensation stack by the rules register and pass. Section 6.10 already introduced a rule that produces the Revert instruction which is handled by the rules revert and revert finished. These rules take one reg-

istered compensation after another and produce the corresponding Exec instructions for their execution.

6.11 Example

The exchange shown in fig. 5.7 can be constructed with the D-Scoop semantics. Deriving the whole execution of the example would be overly long and mostly repetitive. Instead, we focus on two interesting portions, which are marked in the figure.

For easier reading, we used the symbols A_M, R_M, R_Y, R_G, A_Y and A_G instead of natural numbers as the identifiers of: the magenta agent, the magenta register, the yellow register, the green register, the yellow account and the green account processors. For most of the steps, we will omit cells that are not affected by the rule, and the extraction/hiding rules.

First exchange

The processors involved in the first excerpt of the exchange are the magenta agent, the magenta register and the yellow register. The current state of the involved processors is as follows:

$$
\left\langle
\begin{array}{c}
\{A_M, R_M, R_Y, R_G, A_Y, A_G\}_{\text{Ps}} \\
\langle A_M \rangle_{\text{Pi}} \\
\left[
\begin{array}{c}
\left\langle
\begin{array}{c}
\langle 4 \rangle_{\text{Fi}} \left[\mathfrak{L}\right]_{\text{T}} \left[\text{SCall } A_M, \{R_Y\}; \Omega; \upsilon\right]_{\text{IL}} \\
\left[\epsilon\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle \emptyset \rangle_{\text{Is}} \{(R_M, 7)\}_{\text{As}}
\end{array}
\right\rangle_{\text{F}}; \\
\langle \langle 3 \rangle_{\text{Fi}} \left[\mathfrak{L}\right]_{\text{T}} \left[\Omega; \upsilon\right]_{\text{IL}} \left[\epsilon\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle \emptyset \rangle_{\text{Is}} \langle \emptyset \rangle_{\text{As}} \rangle_{\text{F}}; \\
\langle \langle 2 \rangle_{\text{Fi}} \left[\mathfrak{S}\right]_{\text{T}} \left[\Omega; \upsilon\right]_{\text{IL}} \left[\epsilon\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle \emptyset \rangle_{\text{Is}} \langle \emptyset \rangle_{\text{As}} \rangle_{\text{F}}
\end{array}
\right]_{\text{EL}} \\
\left[\mathfrak{N}\right]_{\text{W}} \left[\epsilon\right]_{\text{PL}} \left[\epsilon\right]_{\text{IB}} \left[\epsilon\right]_{\text{OB}} \left[\epsilon\right]_{\text{Ns}}
\end{array}
\right\rangle_{\text{P}}
$$

$$
\left\langle
\begin{array}{c}
\langle R_M \rangle_{\text{Pi}} \\
\left[\langle \langle 7 \rangle_{\text{Fi}} \left[\mathfrak{S}\right]_{\text{T}} \left[\epsilon\right]_{\text{IL}} \left[A_M\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle \emptyset \rangle_{\text{As}} \langle \emptyset \rangle_{\text{Is}} \rangle_{\text{F}}\right]_{\text{EL}} \\
\left[\mathfrak{N}\right]_{\text{W}} \left[\epsilon\right]_{\text{PL}} \left[\epsilon\right]_{\text{IB}} \left[\epsilon\right]_{\text{OB}} \left[\epsilon\right]_{\text{Ns}}
\end{array}
\right\rangle_{\text{P}}
$$

$$
\left\langle
\begin{array}{c}
\langle R_Y \rangle_{\text{Pi}} \\
\left[\epsilon\right]_{\text{EL}} \\
\left[\mathfrak{N}\right]_{\text{W}} \left[\epsilon\right]_{\text{PL}} \left[\epsilon\right]_{\text{IB}} \left[\epsilon\right]_{\text{OB}} \left[\epsilon\right]_{\text{Ns}}
\end{array}
\right\rangle_{\text{P}}
$$

We can see that the agent has three frames. The original frame and two frames from the non-separate calls, one for `transfer` and one for `find_account`. The topmost frame has a lock on the magenta register processor's frame number 7. This frame's owner and client is the agent processor, and its instruction list is empty.

The code to be executed is the **`separate`** block in `find_account`. A **`separate`** block is effectively a non-separate call with access to the local stack variables. As stated before, local variables are not in the scope of the D-Scoop semantics, so a **`separate`** block is treated like a regular non-separate call. This is represented by the SCall $A_M, \{R_Y\}$ instruction at the head of the instruction list of frame 4.

A local call is handled by scall 1, which adds a new frame for the execution of the feature.

$$\frac{\text{fresh}\,(11)}{\begin{array}{c}\langle A_M\rangle_{\text{PI}} \vdash \\ \left[\left\langle\left[\text{SCALL}\ A_M, \{R_Y\}\ldots\right]_{\text{IL}}\langle\emptyset\rangle_{\text{IS}}\langle(R_M,7)\rangle_{\text{AS}}\right\rangle_{\text{F}}\cdots\right]_{\text{EL}} \\ \rightarrow \\ \left[\begin{array}{c}\left\langle\begin{array}{c}\langle 11\rangle_{\text{FI}}\langle\mathfrak{L}\rangle_{\text{T}}\left[\text{EXEC}\ \{R_Y\}, \emptyset, \emptyset; v\right]_{\text{IL}} \\ [\epsilon]_{\text{OL}}\left[\left([\epsilon], A_M\right)\right]_{\text{CL}}\langle(R_M,7)\rangle_{\text{IS}}\langle\emptyset\rangle_{\text{AS}}\end{array}\right\rangle_{\text{F}}; \\ \left\langle[\ldots]_{\text{IL}}\langle\emptyset\rangle_{\text{IS}}\langle(R_M,7)\rangle_{\text{AS}}\right\rangle_{\text{F}}\cdots\end{array}\right]_{\text{EL}}\end{array}}\ \text{SCALL 1}$$

As we can see, the application of the call passes the acquired lock on the magenta register to the new frame. For the execution, a lock on the yellow register processor needs to be acquired. To achieve this, the agent processor, according to the PRELOCK rule, sends a Prelock message.

$$\frac{\{R_Y\} \setminus \emptyset = \{R_Y\} \cup \emptyset \qquad \forall x \in \emptyset : R_Y < x}{\begin{array}{c}\langle A_M\rangle_{\text{PI}} \vdash \\ \left[\left\langle\left[\text{EXEC}\ \{R_Y\}, \emptyset, \emptyset\ldots\right]_{\text{IL}}\right\rangle_{\text{F}}\cdots\right]_{\text{EL}}\left[\ldots\right]_{\text{OB}} \\ \rightarrow \\ \left[\left\langle\left[12 \downarrow R_Y; \text{EXEC}\ \{R_Y\}, \emptyset, \{R_Y\}\ldots\right]_{\text{IL}}\right\rangle_{\text{F}}\cdots\right]_{\text{EL}} \\ \left[\ldots\boxed{\text{Prelock}\ A_M}_{12} \uparrow R_Y\right]_{\text{OB}}\end{array}}\ \text{PRELOCK}$$

The frame is blocked due to the $12 \downarrow R_Y$ instruction, which waits for the OK message with the identifier 12. The message is moved to the other processor through TRANSMIT.

$$\frac{}{\begin{array}{c}\left\langle\left[\boxed{\text{Prelock}\ A_M}_{12} \uparrow R_Y\ldots\right]_{\text{OB}}\right\rangle_{\text{P}}\left\langle\langle R_Y\rangle_{\text{PI}}\left[\ldots\right]_{\text{IB}}\right\rangle_{\text{P}} \\ \rightarrow \\ \left\langle[\ldots]_{\text{OB}}\right\rangle_{\text{P}}\left\langle\langle R_Y\rangle_{\text{PI}}\left[\ldots\boxed{\text{Prelock}\ A_M}_{12}\right]_{\text{IB}}\right\rangle_{\text{P}}\end{array}}\ \text{TRANSMIT}$$

The register now enqueues the prelock request in the prelock list.

$$\frac{}{[\ldots]_{\text{PL}}\left[\boxed{\text{Prelock}\ A_M}_1 2\ldots\right]_{\text{IB}} \rightarrow \left[\ldots(A_M, 12)\right]_{\text{PL}}\left[\ldots\right]_{\text{IB}}}\ \text{PRELOCK ENQUEUE}$$

Since the prelock list was empty before, the register can immediately send the OK message the agent is waiting for.

$$\frac{}{\begin{array}{c}\left[(A_M, 12)\ldots\right]_{\text{PL}}\left[\ldots\right]_{\text{OB}}\langle\emptyset\rangle_{\text{NS}} \rightarrow \\ \left[A_M\ldots\right]_{\text{PL}}\left[\boxed{\text{OK}}_1 2 \uparrow A_M\ldots\right]_{\text{OB}}\langle\emptyset\rangle_{\text{NS}}\end{array}}\ \text{PRELOCK REPLY}$$

The next step is to transmit the reply, but we skip it for brevity. The agent can now unblock the instruction list by applying RECV. OK.

$$\frac{}{\left[\ldots\left\langle\left[12 \downarrow R_Y\ldots\right]_{\text{IL}}\right\rangle_{\text{F}}\cdots\right]_{\text{EL}}\left[\boxed{\text{OK}}_{12}\cdots\right]_{\text{IB}} \rightarrow \left[\ldots\left\langle\left[\ldots\right]_{\text{IL}}\right\rangle_{\text{F}}\cdots\right]_{\text{EL}}\left[\ldots\right]_{\text{IB}}}\ \text{RECV. OK}$$

The last argument of the EXEC instruction matches the first; this means that the prelocking phase is over and the locking phase starts. Accordingly, the agent sends the Lock request.

$$\frac{\begin{array}{c}\{R_Y\} = \{R_Y\} \cup \emptyset \\ \text{fresh}\,(13)\end{array}}{\begin{array}{c}\langle A_M \rangle_{\text{PI}} \vdash \\ \left[\langle \left[\textsc{Exec}\ \{R_Y\}, \emptyset, \{R_Y\} \ldots\right]_{\text{IL}} \langle \emptyset \rangle_{\text{AS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \left[\ldots\right]_{\text{OB}} \\ \rightarrow \\ \left[\langle \left[13 \downarrow R_M ; \textsc{Exec}\ \{R_Y\}, \emptyset, \{R_Y\} \ldots\right]_{\text{IL}} \langle \emptyset \rangle_{\text{AS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \\ \left[\ldots \boxed{\text{Lock}\ A_M}_{13} \uparrow R_M\right]_{\text{OB}}\end{array}}\ \textsc{lock}$$

We skip the TRANSMIT step again and immediately continue with LOCK REPLY.

$$\frac{\text{fresh}\,(14)}{\begin{array}{c}\langle R_Y \rangle_{\text{PI}} \vdash \\ \left[\ldots\right]_{\text{EL}} \left[A_M \ldots\right]_{\text{PL}} \left[\boxed{\text{Lock}\ A_M}_{13}\right]_{\text{IB}} \left[\ldots\right]_{\text{OB}} \rightarrow \\ \left[\langle \langle 14 \rangle_{\text{FI}} \langle \mathfrak{S} \rangle_{\text{T}} \left[\epsilon\right]_{\text{IL}} \left[A_M\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle \emptyset \rangle_{\text{AS}} \langle \emptyset \rangle_{\text{IS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \\ \left[\ldots\right]_{\text{PL}} \left[\ldots\right]_{\text{IB}} \left[\ldots \boxed{\text{OK}\,(R_Y, 14)}_{13}\right]_{\text{OB}}\end{array}}\ \textsc{lock reply}$$

Another skipped transmission and the agent is ready to add the new lock to its set of acquired locks. But the value first needs to move from the in-box to the instruction list.

$$\frac{}{\begin{array}{c}\left[\ldots \langle \left[13 \downarrow R_Y \ldots\right]_{\text{IL}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \left[\boxed{\text{OK}\,(R_M, 14)}_{13} \ldots\right]_{\text{IB}} \\ \rightarrow \\ \left[\ldots \langle \left[\downarrow (R_Y, 14) \ldots\right]_{\text{IL}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \left[\ldots\right]_{\text{IB}}\end{array}}\ \textsc{recv. value}$$

$$\frac{}{\begin{array}{c}\left[\langle \left[\downarrow (R_Y, 14) ; \textsc{Exec}\ \{R_Y\}, \emptyset, \{R_Y\} \ldots\right]_{\text{IL}} \langle \emptyset \rangle_{\text{AS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \\ \rightarrow \\ \left[\langle \langle 4 \rangle_{\text{FI}} \left[\textsc{Exec}\ \{R_Y\}, \emptyset, \{R_Y\} \ldots\right]_{\text{IL}} \left\{(R_Y, 14)\right\}_{\text{AS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}}\end{array}}\ \textsc{add lock}$$

With all the required locks in place, the execution of the separate block can start and the EXEC instruction can finally be consumed.

$$\frac{\{R_Y\} \subset \{R_Y\}}{\begin{array}{c}\left[\langle \left[\textsc{Exec}\ \{R_Y\}, \emptyset, \{R_Y\} \ldots\right]_{\text{IL}} \langle (R_Y, 14) \rangle_{\text{AS}} \langle (R_M, 7) \rangle_{\text{IS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}} \\ \rightarrow \\ \left[\langle \left[\Omega \ldots\right]_{\text{IL}} \langle (R_Y, 14) \rangle_{\text{AS}} \left\{(R_M, 7)\right\}_{\text{IS}} \rangle_{\text{F}} \ldots\right]_{\text{EL}}\end{array}}\ \textsc{exec}$$

The state of the involved processors is now:

$$\left\langle \begin{array}{c} \langle A_M \rangle_{\text{PI}} \\ \left[\begin{array}{c} \left\langle \begin{array}{c} \langle 11 \rangle_{\text{FI}} \left[\mathfrak{L}\right]_{\text{T}} \left[\Omega; \upsilon\right]_{\text{IL}} \\ \left[\epsilon\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \left\{\left(R_M, 7\right)\right\}_{\text{IS}} \left\{\left(R_Y, 14\right)\right\}_{\text{AS}} \end{array} \right\rangle_{\text{F}}; \\ \left\langle \begin{array}{c} \langle 4 \rangle_{\text{FI}} \left[\mathfrak{L}\right]_{\text{T}} \left[\Omega; \upsilon\right]_{\text{IL}} \\ \left[\epsilon\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle\emptyset\rangle_{\text{IS}} \left\{\left(R_M, 7\right)\right\}_{\text{AS}} \end{array} \right\rangle_{\text{F}}; \\ \left\langle \langle 3 \rangle_{\text{FI}} \left[\mathfrak{L}\right]_{\text{T}} \left[\Omega; \upsilon\right]_{\text{IL}} \left[\epsilon\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle\emptyset\rangle_{\text{IS}} \langle\emptyset\rangle_{\text{AS}} \right\rangle_{\text{F}}; \\ \left\langle \langle 2 \rangle_{\text{FI}} \left[\mathfrak{L}\right]_{\text{T}} \left[\Omega; \upsilon\right]_{\text{IL}} \left[\epsilon\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle\emptyset\rangle_{\text{IS}} \langle\emptyset\rangle_{\text{AS}} \right\rangle_{\text{F}} \end{array} \right]_{\text{EL}} \\ \left[\epsilon\right]_{\text{PL}} \left[\epsilon\right]_{\text{IB}} \left[\epsilon\right]_{\text{OB}} \left[\epsilon\right]_{\text{NS}} \end{array} \right\rangle_{\text{P}}$$

$$\left\langle \begin{array}{c} \langle R_M \rangle_{\text{PI}} \\ \left[\langle \langle 7 \rangle_{\text{FI}} \left[\mathfrak{S}\right]_{\text{T}} \left[\epsilon\right]_{\text{IL}} \left[A_M\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle\emptyset\rangle_{\text{AS}} \langle\emptyset\rangle_{\text{IS}} \rangle_{\text{F}}\right]_{\text{EL}} \\ \left[\epsilon\right]_{\text{PL}} \left[\epsilon\right]_{\text{IB}} \left[\epsilon\right]_{\text{OB}} \left[\epsilon\right]_{\text{NS}} \end{array} \right\rangle_{\text{P}}$$

$$\left\langle \begin{array}{c} \langle R_Y \rangle_{\text{PI}} \\ \left[\langle \langle 14 \rangle_{\text{FI}} \left[\mathfrak{S}\right]_{\text{T}} \left[\epsilon\right]_{\text{IL}} \left[A_M\right]_{\text{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \langle\emptyset\rangle_{\text{AS}} \langle\emptyset\rangle_{\text{IS}} \rangle_{\text{F}}\right]_{\text{EL}} \\ \left[\epsilon\right]_{\text{PL}} \left[\epsilon\right]_{\text{IB}} \left[\epsilon\right]_{\text{OB}} \left[\epsilon\right]_{\text{NS}} \end{array} \right\rangle_{\text{P}}$$

The agent processor now has an additional frame, 11, which has an *inherited* lock on the magenta register and an *acquired* lock on the yellow register. The inherited lock is due to the non-separate call rule, which copies all acquired locks of the previous frame to the inherited lock set of the new frame. The new frame is about to call `lookup_account` of the yellow register, so we apply NEXT SCALL 2:

$$\frac{\begin{array}{c} \left(R_Y, 14\right) \in \left\{\left(R_Y, 14\right), \left(R_M, 7\right)\right\} \\ \left\{\left(R_M, 7\right)\right\} = \left\{\left(R_Y, 14\right), \left(R_M, 7\right)\right\} \setminus \left\{\left(R_Y, 14\right)\right\} \\ \emptyset \subset \left\{A_M, R_M, R_Y, R_G, A_Y, A_G\right\} \end{array}}{\begin{array}{c} \left\{A_M, R_M, R_Y, R_G, A_Y, A_G\right\}_{\text{PS}} \langle A_M \rangle_{\text{PI}} \vdash \\ \left[\langle \left[\Omega \ldots\right]_{\text{IL}} \left\{\left(R_Y, 14\right)\right\}_{\text{AS}} \left\{\left(R_M, 7\right)\right\}_{\text{IS}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \\ \rightarrow \\ \left[\left\langle \begin{array}{c} \left[\text{PASS } \left\{\left(R_M, 7\right)\right\}, R_Y; \text{SCALL } R_Y, \emptyset; \Omega \ldots\right]_{\text{IL}} \\ \left\{\left(R_Y, 14\right)\right\}_{\text{AS}} \left\{\left(R_M, 7\right)\right\}_{\text{IS}} \end{array} \right\rangle_{\text{F}} \cdots\right]_{\text{EL}} \end{array}} \text{ NEXT SCALL 2}$$

For execution of the `lookup_account` feature of the *yellow* register, the (inherited) lock on the *magenta* register will be passed on. This is because the call is synchronous and the agent still holds the lock of the magenta register. Had the feature not had the register as an argument, and instead used another **`separate`** block that was closed before the second call, then it would have released the lock on the magenta processor before and not cause lock passing. So instead of immediately discarding the PASS instruction, we apply PASS first:

$$\frac{\text{fresh}\left(15\right)}{\begin{array}{c} \langle A_M \rangle_{\text{PI}} \vdash \\ \left[\langle \left[\text{PASS } \left\{\left(R_M, 7\right)\right\} \cup \emptyset, R_Y \ldots\right]_{\text{IL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \left[\cdots\right]_{\text{OB}} \\ \rightarrow \\ \left[\langle \left[\text{PASS } \emptyset, R_Y \ldots\right]_{\text{IL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \left[\cdots \boxed{\text{PASS } A_M, 7, R_Y}_{15} \uparrow R_M\right]_{\text{OB}} \end{array}} \text{ PASS}$$

The processor signaled the passing of a lock, so we remove the instruction with PASS FIN.

$$\frac{}{\left[\left\langle\left[\textsc{Pass}\ \emptyset, R_Y \ldots\right]_{\textsc{Il}}\right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}} \rightarrow \left[\left\langle\left[\ldots\right]_{\textsc{Il}}\right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}}}\ \textsc{pass fin.}$$

The call message, produced by SCALL 2, includes both the lock on the magenta register as well as a new lock on the agent processor, on a newly created frame:

$$\frac{\begin{array}{c} fresh(16,17) \\ (R_Y, 14) \in \{(R_M, 7), (R_Y, 14)\} \\ \{(R_M, 7), (A_M, 16)\} = \{(R_M, 7), (R_Y, 14), (A_M, 16)\} \setminus \{(R_Y, 14)\} \end{array}}{\begin{array}{c} \langle A_M \rangle_{\textsc{Pi}} \vdash \\ \left[\left\langle\left[\textsc{SCall}\ R_Y, \emptyset \ldots\right]_{\textsc{Il}} \left\{(R_M, 7)\right\}_{\textsc{Is}} \left\{(R_Y, 14)\right\}_{\textsc{As}}\right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}} \left[\ldots\right]_{\textsc{Ob}} \\ \rightarrow \\ \left[\begin{array}{c} \left\langle\left\langle 16\right\rangle_{\textsc{Fi}}\left\langle \mathfrak{C}\right\rangle_{\textsc{T}} \left[\pi_{R_Y}\right]_{\textsc{Il}} \left[R_Y\right]_{\textsc{Ol}} \left[\epsilon\right]_{\textsc{Cl}} \left\langle\emptyset\right\rangle_{\textsc{Is}} \left\langle\emptyset\right\rangle_{\textsc{As}}\right\rangle_{\textsc{F}}; \\ \left\langle\left[17 \downarrow R_Y \ldots\right]_{\textsc{Il}} \left\{(R_M, 7)\right\}_{\textsc{Is}} \left\{(R_Y, 14)\right\}_{\textsc{As}}\right\rangle_{\textsc{F}} \cdots \end{array}\right]_{\textsc{El}} \\ \left[\ldots \boxed{\textsc{SCall}\ A_M, 14, \emptyset, \{(R_M, 7), (A_M, 16)\}}_{17} \uparrow R_Y\right]_{\textsc{Ob}} \end{array}}\ \textsc{scall 2}$$

After the transmission of the two messages in the out-box of the agent processor, the involved processors are in the following state:

$$\left\langle\begin{array}{c} \langle A_M \rangle_{\textsc{Pi}} \\ \left[\begin{array}{c} \left\langle\left\langle 16\right\rangle_{\textsc{Fi}}\left\langle \mathfrak{C}\right\rangle_{\textsc{T}} \left[\pi_{R_Y}\right]_{\textsc{Il}} \left[R_Y\right]_{\textsc{Ol}} \left[\epsilon\right]_{\textsc{Cl}} \left\langle\emptyset\right\rangle_{\textsc{Is}} \left\langle\emptyset\right\rangle_{\textsc{As}}\right\rangle_{\textsc{F}}; \\ \left\langle\left\langle 11\right\rangle_{\textsc{Fi}}\left\langle \mathfrak{L}\right\rangle_{\textsc{T}} \left[17 \downarrow R_Y; \Omega; \upsilon\right]_{\textsc{Il}} \left[\epsilon\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\{(R_M, 7)\right\}_{\textsc{Is}} \left\{(R_Y, 14)\right\}_{\textsc{As}}\right\rangle_{\textsc{F}}; \\ \left\langle\left\langle 4\right\rangle_{\textsc{Fi}}\left\langle \mathfrak{L}\right\rangle_{\textsc{T}} \left[\Omega; \upsilon\right]_{\textsc{Il}} \left[\epsilon\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle\emptyset\right\rangle_{\textsc{Is}} \left\{(R_M, 7)\right\}_{\textsc{As}}\right\rangle_{\textsc{F}}; \\ \left\langle\left\langle 3\right\rangle_{\textsc{Fi}}\left\langle \mathfrak{L}\right\rangle_{\textsc{T}} \left[\Omega; \upsilon\right]_{\textsc{Il}} \left[\epsilon\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle\emptyset\right\rangle_{\textsc{Is}} \left\langle\emptyset\right\rangle_{\textsc{As}}\right\rangle_{\textsc{F}}; \\ \left\langle\left\langle 2\right\rangle_{\textsc{Fi}}\left\langle \mathfrak{S}\right\rangle_{\textsc{T}} \left[\Omega; \upsilon\right]_{\textsc{Il}} \left[\epsilon\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle\emptyset\right\rangle_{\textsc{Is}} \left\langle\emptyset\right\rangle_{\textsc{As}}\right\rangle_{\textsc{F}} \end{array}\right]_{\textsc{El}} \\ \left[\epsilon\right]_{\textsc{Pl}} \left[\epsilon\right]_{\textsc{Ib}} \left[\epsilon\right]_{\textsc{Ob}} \left[\epsilon\right]_{\textsc{Ns}} \end{array}\right\rangle_{\textsc{P}}$$

$$\left\langle\begin{array}{c} \langle R_M \rangle_{\textsc{Pi}} \\ \left[\left\langle\left\langle 7\right\rangle_{\textsc{Fi}}\left\langle \mathfrak{S}\right\rangle_{\textsc{T}} \left[\epsilon\right]_{\textsc{Il}} \left[A_M\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle\emptyset\right\rangle_{\textsc{As}} \left\langle\emptyset\right\rangle_{\textsc{Is}}\right\rangle_{\textsc{F}}\right]_{\textsc{El}} \\ \left[\epsilon\right]_{\textsc{Pl}} \left[\boxed{\textsc{Pass}\ A_M, 7, R_Y}_{15}\right]_{\textsc{Ib}} \left[\epsilon\right]_{\textsc{Ob}} \left[\epsilon\right]_{\textsc{Ns}} \end{array}\right\rangle_{\textsc{P}}$$

$$\left\langle\begin{array}{c} \langle R_Y \rangle_{\textsc{Pi}} \\ \left[\left\langle\left\langle 14\right\rangle_{\textsc{Fi}}\left\langle \mathfrak{S}\right\rangle_{\textsc{T}} \left[\epsilon\right]_{\textsc{Il}} \left[A_M\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle\emptyset\right\rangle_{\textsc{As}} \left\langle\emptyset\right\rangle_{\textsc{Is}}\right\rangle_{\textsc{F}}\right]_{\textsc{El}} \\ \left[\epsilon\right]_{\textsc{Pl}} \left[\boxed{\textsc{SCall}\ A_M, 14, \emptyset, \{(R_M, 7), (A_M, 16)\}}_{17}\right]_{\textsc{Ib}} \left[\epsilon\right]_{\textsc{Ob}} \left[\epsilon\right]_{\textsc{Ns}} \end{array}\right\rangle_{\textsc{P}}$$

Even though the magenta register processor has not yet processed the $\boxed{\textsc{Pass}}$ message, the yellow register processor can start application of the call with ENQ. SCALL.

$$\frac{\emptyset = \emptyset \setminus \{R_M, A_M, R_Y\}}{\begin{array}{c} \langle R_Y \rangle_{\textsc{Pi}} \vdash \\ \left[\ldots\left\langle\left\langle 14\right\rangle_{\textsc{Fi}} \left[\ldots\right]_{\textsc{Il}} \left[A_M \ldots\right]_{\textsc{Ol}}\right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}} \\ \left[\boxed{\textsc{SCall}\ A_M, 14, \emptyset, \{(R_M, 7), (A_M, 16)\}}_{17} \cdots\right]_{\textsc{Ib}} \\ \rightarrow \\ \left[\ldots\left\langle\begin{array}{c} \langle 14 \rangle_{\textsc{Fi}} \\ \left[\ldots \textsc{Exec}\ \emptyset, \{(R_M, 7), (A_M, 16)\}, \emptyset; \right]_{\textsc{Il}} \left[A_M \ldots\right]_{\textsc{Ol}} \\ \textsc{Return}\ A_M, 17 \end{array}\right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}} \\ \left[\ldots\right]_{\textsc{Ib}} \end{array}}\ \textsc{enq. scall}$$

Without any separate arguments, there are no additional locks needed. The yellow register can start EXECution:

$$\frac{\emptyset \subseteq \{R_M, A_M\}}{\begin{array}{c}\left[\left\langle\left[\textsc{Exec}\ \emptyset, \{(R_M, 7), (A_M, 16)\}, \emptyset \ldots\right]_{\textsc{Il}} \langle\emptyset\rangle_{\textsc{Is}} \langle\emptyset\rangle_{\textsc{As}}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}} \\ \rightarrow \\ \left[\left\langle\left[\Omega \ldots\right]_{\textsc{Il}} \langle\emptyset\rangle_{\textsc{Is}} \langle\emptyset\rangle_{\textsc{As}}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}}\end{array}}\ \textsc{exec}$$

The magenta register is making some non-separate calls to look up the correct value. We omit them — they produce no messages and the state is the same afterwards — and instead let the Ω of the register produce an UNLOCK instruction.

$$\frac{}{\left[\left\langle\left[\Omega \ldots\right]_{\textsc{Il}}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}} \rightarrow \left[\left\langle\left[\textsc{Unlock} \ldots\right]_{\textsc{Il}}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}}}\ \textsc{next unlock}$$

The UNLOCK instruction is sending an $\boxed{\text{Unlock}}$ message to the agent processor through application of UNLOCK 2.

$$\frac{\begin{array}{c}R_Y \neq A_M \\ \text{fresh}\,(18)\end{array}}{\begin{array}{c}\langle R_Y\rangle_{\textsc{Pi}} \vdash \\ \left[\left\langle\begin{array}{c}\left[\textsc{Unlock} \ldots\right]_{\textsc{Il}} \\ \langle\{(A_M, 16)\} \cup \{(R_M, 7)\}\rangle_{\textsc{Is}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\textsc{Cl}}\end{array}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}} \\ \left[\cdots\right]_{\textsc{Ob}} \\ \rightarrow \\ \left[\left\langle\begin{array}{c}\left[\textsc{Unlock} \ldots\right]_{\textsc{Il}} \\ \langle\{(R_M, 7)\}\rangle_{\textsc{Is}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\textsc{Cl}}\end{array}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}} \\ \left[\ldots \boxed{\text{Unlock } R_Y, 16}_{18} \uparrow A_M\right]_{\textsc{Ob}}\end{array}}\ \textsc{unlock 2}$$

But since it also got passed a lock on the magenta register, it also needs to unlock this one.

$$\frac{\begin{array}{c}R_Y \neq A_M \\ \text{fresh}\,(19)\end{array}}{\begin{array}{c}\langle R_Y\rangle_{\textsc{Pi}} \vdash \\ \left[\left\langle\begin{array}{c}\left[\textsc{Unlock} \ldots\right]_{\textsc{Il}} \\ \langle\{(R_M, 7)\} \cup \emptyset\rangle_{\textsc{Is}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\textsc{Cl}}\end{array}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}} \\ \left[\cdots\right]_{\textsc{Ob}} \\ \rightarrow \\ \left[\left\langle\begin{array}{c}\left[\textsc{Unlock} \ldots\right]_{\textsc{Il}} \\ \langle\emptyset\rangle_{\textsc{Is}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\textsc{Cl}}\end{array}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}} \\ \left[\ldots \boxed{\text{Unlock } R_Y, 7}_{19} \uparrow R_M\right]_{\textsc{Ob}}\end{array}}\ \textsc{unlock 2}$$

Now the UNLOCK instruction can be removed through UNLOCK FINISHED

$$\frac{\emptyset = \emptyset \vee R_Y = A_M}{\begin{array}{c}\langle R_Y\rangle_{\textsc{Pi}} \vdash \\ \left[\left\langle\left[\textsc{Unlock} \ldots\right]_{\textsc{Il}} \langle\emptyset\rangle_{\textsc{As}} \langle\emptyset\rangle_{\textsc{Is}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\textsc{Cl}}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}} \\ \rightarrow \\ \left[\left\langle\left[\ldots\right]_{\textsc{Il}} \langle\emptyset\rangle_{\textsc{As}} \langle\emptyset\rangle_{\textsc{Is}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\textsc{Cl}}\right\rangle_{\textsc{F}} \cdots\right]_{\textsc{El}}\end{array}}\ \textsc{unlock finished}$$

Then the yellow register sends the result to the magenta agent by processing the RETURN instruction through RETURN.

$$\frac{}{\begin{array}{c}\left\langle \left[\textsc{Return}\ A_M, 17 \ldots\right]_{\textsc{Il}} \right\rangle_{\textsc{F}} \left[\ldots\right]_{\textsc{Ob}} \\ \rightarrow \\ \left\langle \left[\ldots\right]_{\textsc{Il}} \right\rangle_{\textsc{F}} \left[\ldots \boxed{\text{OK}}_{17} \uparrow A_M\right]_{\textsc{Ob}}\end{array}}\ \textsc{return}$$

After a triple application of TRANSMIT, the state of the two processors is now:

$$\left\langle \begin{array}{c} \left\langle A_M \right\rangle_{\textsc{Pi}} \\ \left[\begin{array}{c} \left\langle \left\langle 16 \right\rangle_{\textsc{Fi}} \left[\pi_{R_Y}\right]_{\textsc{Il}} \left[R_Y\right]_{\textsc{Ol}} \left[\epsilon\right]_{\textsc{Cl}} \left\langle \emptyset \right\rangle_{\textsc{Is}} \left\langle \emptyset \right\rangle_{\textsc{As}} \right\rangle_{\textsc{F}}; \\ \left\langle \left\langle 11 \right\rangle_{\textsc{Fi}} \left[17 \downarrow R_Y; \Omega; \upsilon\right]_{\textsc{Il}} \right. \\ \left. \left[\epsilon\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\{\left(R_M, 7\right)\right\}_{\textsc{Is}} \left\{\left(R_Y, 14\right)\right\}_{\textsc{As}} \right\rangle_{\textsc{F}}; \\ \left\langle \left\langle 4 \right\rangle_{\textsc{Fi}} \left[\Omega; \upsilon\right]_{\textsc{Il}} \left[\epsilon\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle \emptyset \right\rangle_{\textsc{Is}} \left\{\left(R_M, 7\right)\right\}_{\textsc{As}} \right\rangle_{\textsc{F}}; \\ \left\langle \left\langle 3 \right\rangle_{\textsc{Fi}} \left[\Omega; \upsilon\right]_{\textsc{Il}} \left[\epsilon\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle \emptyset \right\rangle_{\textsc{Is}} \left\langle \emptyset \right\rangle_{\textsc{As}} \right\rangle_{\textsc{F}}; \\ \left\langle \left\langle 2 \right\rangle_{\textsc{Fi}} \left[\Omega; \upsilon\right]_{\textsc{Il}} \left[\epsilon\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle \emptyset \right\rangle_{\textsc{Is}} \left\langle \emptyset \right\rangle_{\textsc{As}} \right\rangle_{\textsc{F}} \\ \left[\epsilon\right]_{\textsc{Pl}} \left[\boxed{\text{Unlock } R_Y, 16}_{18}; \boxed{\text{OK}}_{17}\right]_{\textsc{Ib}} \left[\epsilon\right]_{\textsc{Ob}} \left[\epsilon\right]_{\textsc{Ns}} \end{array} \right]_{\textsc{El}} \end{array} \right\rangle_{\textsc{P}}$$

$$\left\langle \begin{array}{c} \left\langle R_M \right\rangle_{\textsc{Pi}} \\ \left[\left\langle \left\langle 7 \right\rangle_{\textsc{Fi}} \left[\epsilon\right]_{\textsc{Il}} \left[A_M\right]_{\textsc{Ol}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle \emptyset \right\rangle_{\textsc{As}} \left\langle \emptyset \right\rangle_{\textsc{Is}} \right\rangle_{\textsc{F}} \right]_{\textsc{El}} \\ \left[\epsilon\right]_{\textsc{Pl}} \left[\boxed{\text{Pass } A_M, 7, R_Y}_{15}; \boxed{\text{Unlock } R_Y, 7}_{19}\right]_{\textsc{Ib}} \left[\epsilon\right]_{\textsc{Ob}} \left[\epsilon\right]_{\textsc{Ns}} \end{array} \right\rangle_{\textsc{P}}$$

$$\left\langle \begin{array}{c} \left\langle R_Y \right\rangle_{\textsc{Pi}} \\ \left[\left\langle \begin{array}{c} \left\langle 14 \right\rangle_{\textsc{Fi}} \left[\epsilon\right]_{\textsc{Il}} \left[A_M\right]_{\textsc{Ol}} \\ \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \left\langle \emptyset \right\rangle_{\textsc{As}} \left\langle \left\{\left(R_M, 7\right), \left(A_M, 16\right)\right\} \right\rangle_{\textsc{Is}} \end{array} \right\rangle_{\textsc{F}} \right]_{\textsc{El}} \\ \left[\epsilon\right]_{\textsc{Pl}} \left[\epsilon\right]_{\textsc{Ib}} \left[\epsilon\right]_{\textsc{Ob}} \left[\epsilon\right]_{\textsc{Ns}} \end{array} \right\rangle_{\textsc{P}}$$

The agent processor can process the $\boxed{\text{Unlock}}$ message through UNLOCK OK.

$$\frac{}{\begin{array}{c}\left[\ldots \left\langle \left\langle 16 \right\rangle_{\textsc{Fi}} \left[\ldots\right]_{\textsc{Il}} \left[R_Y \ldots\right]_{\textsc{Ol}} \right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}} \left[\boxed{\text{Unlock } R_Y, 16}_{18} \ldots\right]_{\textsc{Ib}} \\ \rightarrow \\ \left[\ldots \left\langle \left\langle 16 \right\rangle_{\textsc{Fi}} \left[\ldots \upsilon\right]_{\textsc{Il}} \left[\ldots\right]_{\textsc{Ol}} \right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}} \left[\ldots\right]_{\textsc{Ib}}\end{array}}\ \textsc{unlock ok}$$

After unblocking the second frame with an application of RECV. OK, which we do not show here, the agent processor consumes the υ using POP CLIENT 3, since the frame was for separate callbacks.

$$\frac{}{\begin{array}{c}\left\langle A_M \right\rangle_{\textsc{Pi}} \vdash \\ \left[\left\langle \left\langle \mathfrak{C} \right\rangle_{\textsc{T}} \left[\upsilon \ldots\right]_{\textsc{Il}} \left[\left(\left[\epsilon\right], R_Y\right)\right]_{\textsc{Cl}} \right\rangle_{\textsc{F}}; \left\langle \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}} \right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}} \\ \rightarrow \\ \left[\left\langle \left[\left(\left[\epsilon; \epsilon\right], A_M\right)\right]_{\textsc{Cl}} \right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}}\end{array}}\ \textsc{pop client 3}$$

The magenta agent now has the information it needs, so we use NEXT UNLOCK to exit the **separate** block.

$$\frac{}{\left[\left\langle \left[\Omega \ldots\right]_{\textsc{Il}} \right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}} \rightarrow \left[\left\langle \left[\textsc{Unlock} \ldots\right]_{\textsc{Il}} \right\rangle_{\textsc{F}} \ldots\right]_{\textsc{El}}}\ \textsc{next unlock}$$

The processor needs to release the lock on the yellow register.

$$\frac{\begin{array}{c} R_Y \neq A_M \\ \text{fresh}\,(20) \end{array}}{\begin{array}{c} \langle A_M \rangle_{\text{PI}} \vdash \\ \left[\langle\langle 11\rangle_{\text{FI}} \left[\text{UNLOCK}\ldots\right]_{\text{IL}} \left\{(R_Y, 14) \cup \emptyset\right\}_{\text{AS}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \left[\cdots\right]_{\text{OB}} \\ \rightarrow \\ \left[\langle\langle 11\rangle_{\text{FI}} \left[\text{UNLOCK}\ldots\right]_{\text{IL}} \langle\emptyset\rangle_{\text{AS}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \left[\cdots \boxed{\text{Unlock } A_M, 14}_{20} \uparrow R_Y\right]_{\text{OB}} \end{array}}\ \text{UNLOCK 1}$$

It can then remove the UNLOCK instruction with UNLOCK FINISHED and process the υ symbol with POP CLIENT 4, which handles frames for non-separate calls.

$$\frac{\{(R_M, 7)\} = \emptyset \vee A_M = A_M}{\begin{array}{c} \langle A_M \rangle_{\text{PI}} \vdash \\ \left[\langle \left[\text{UNLOCK}\ldots\right]_{\text{IL}} \langle\emptyset\rangle_{\text{AS}} \left\{(R_M, 7)\right\}_{\text{IS}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\text{CL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \\ \rightarrow \\ \left[\langle \left[\cdots\right]_{\text{IL}} \langle\emptyset\rangle_{\text{AS}} \left\{(R_M, 7)\right\}_{\text{IS}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\text{CL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \end{array}}\ \text{UNLOCK FINISHED}$$

$$\frac{}{\begin{array}{c} \langle A_M \rangle_{\text{PI}} \vdash \\ \left[\begin{array}{c} \langle\langle \mathfrak{L}\rangle_{\text{T}} \left[\upsilon; \epsilon\right]_{\text{IL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\text{CL}} \rangle_{\text{F}}; \\ \langle \left[\cdots\right]_{\text{IL}} \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\text{CL}} \rangle_{\text{F}} \cdots \end{array}\right]_{\text{EL}} \\ \rightarrow \\ \left[\langle \left[\epsilon; \ldots\right]_{\text{IL}} \left[\left(\left[\epsilon; \epsilon\right], A_M\right) \ldots\right]_{\text{CL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \end{array}}\ \text{POP CLIENT 4}$$

The magenta processor has two messages waiting in its in-box. It first processes the $\boxed{\text{Pass}}$ message.

$$\frac{}{\begin{array}{c} \left[\langle \left[\cdots\right]_{\text{IL}} \left[A_M \cdots\right]_{\text{OL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \left[\boxed{\text{Pass } A_M, 7, R_Y}_{15} \cdots\right]_{\text{IB}} \\ \rightarrow \\ \left[\langle \left[\ldots \pi_{R_Y}\right]_{\text{IL}} \left[R_Y; A_M \cdots\right]_{\text{OL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \left[\cdots\right]_{\text{IB}} \end{array}}\ \text{PASS OK}$$

It can then immediately add the υ symbol through UNLOCK OK.

$$\frac{}{\begin{array}{c} \left[\langle\langle 7\rangle_{\text{FI}} \left[\cdots\right]_{\text{IL}} \left[R_Y \cdots\right]_{\text{OL}} \rangle_{\text{F}}\right]_{\text{EL}} \left[\boxed{\text{Unlock } R_Y, 7}_{19} \cdots\right]_{\text{IB}} \\ \rightarrow \\ \left[\langle\langle 7\rangle_{\text{FI}} \left[\ldots \upsilon\right]_{\text{IL}} \left[\cdots\right]_{\text{OL}} \rangle_{\text{F}}\right]_{\text{EL}} \left[\cdots\right]_{\text{IB}} \end{array}}\ \text{UNLOCK OK}$$

The magenta register now first pushes the yellow register through PUSH CLIENT.

$$\frac{}{\left[\langle \left[\pi_{R_Y} \cdots\right]_{\text{IL}} \left[\cdots\right]_{\text{CL}} \rangle_{\text{F}}\right]_{\text{EL}} \rightarrow \left[\langle \left[\pi_{R_Y} \cdots\right]_{\text{IL}} \left[\left(\left[\epsilon\right], R_Y\right) \cdots\right]_{\text{CL}} \rangle_{\text{F}}\right]_{\text{EL}}}\ \text{PUSH CLIENT}$$

And then removes it again with pop client 1.

$$\frac{}{\begin{array}{c} \left[\langle \left[\upsilon \ldots\right]_{\text{IL}} \left[\left(\left[\epsilon\right], R_Y\right); \left(\left[\epsilon\right], A_M\right) \ldots\right]_{\text{CL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \\ \rightarrow \\ \left[\langle \left[\cdots\right]_{\text{IL}} \left[\left(\left[\epsilon; \epsilon\right], A_M\right) \ldots\right]_{\text{CL}} \rangle_{\text{F}} \cdots\right]_{\text{EL}} \end{array}}\ \text{POP CLIENT 1}$$

The yellow register can also process its Unlock message through UNLOCK OK and remove the frame with POP CLIENT 2. Afterwards, the system is back in the same state as before the **separate** block.

$$\left\langle \begin{array}{c} \{A_M, R_M, R_Y, R_G, A_Y, A_G\}_{\mathrm{Ps}} \\ \langle A_M \rangle_{\mathrm{PI}} \\ \left[\begin{array}{l} \left\langle \begin{array}{c} \langle 4 \rangle_{\mathrm{FI}} \left[\mathfrak{L}\right]_{\mathrm{T}} \left[\textsc{SCall}\ A_M, \{R_Y\}; \Omega; \upsilon\right]_{\mathrm{IL}} \\ \left[\epsilon\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{Is}} \left\{(R_M, 7)\right\}_{\mathrm{As}} \end{array} \right\rangle_{\mathrm{F}}; \\ \langle\langle 3 \rangle_{\mathrm{FI}} \left[\mathfrak{L}\right]_{\mathrm{T}} \left[\Omega; \upsilon\right]_{\mathrm{IL}} \left[\epsilon\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{Is}} \langle \emptyset \rangle_{\mathrm{As}} \rangle_{\mathrm{F}}; \\ \langle\langle 2 \rangle_{\mathrm{FI}} \left[\mathfrak{S}\right]_{\mathrm{T}} \left[\Omega; \upsilon\right]_{\mathrm{IL}} \left[\epsilon\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{Is}} \langle \emptyset \rangle_{\mathrm{As}} \rangle_{\mathrm{F}} \end{array} \right]_{\mathrm{EL}} \\ \left[\mathfrak{N}\right]_{\mathrm{W}} \left[\epsilon\right]_{\mathrm{PL}} \left[\epsilon\right]_{\mathrm{IB}} \left[\epsilon\right]_{\mathrm{OB}} \left[\epsilon\right]_{\mathrm{Ns}} \end{array} \right\rangle_{\mathrm{P}}$$

$$\left\langle \begin{array}{c} \langle R_M \rangle_{\mathrm{PI}} \\ \left[\langle\langle 7 \rangle_{\mathrm{FI}} \left[\mathfrak{S}\right]_{\mathrm{T}} \left[\epsilon\right]_{\mathrm{IL}} \left[A_M\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{As}} \langle \emptyset \rangle_{\mathrm{Is}} \rangle_{\mathrm{F}}\right]_{\mathrm{EL}} \\ \left[\mathfrak{N}\right]_{\mathrm{W}} \left[\epsilon\right]_{\mathrm{PL}} \left[\epsilon\right]_{\mathrm{IB}} \left[\epsilon\right]_{\mathrm{OB}} \left[\epsilon\right]_{\mathrm{Ns}} \end{array} \right\rangle_{\mathrm{P}}$$

$$\left\langle \begin{array}{c} \langle R_Y \rangle_{\mathrm{PI}} \\ \left[\epsilon\right]_{\mathrm{EL}} \\ \left[\mathfrak{N}\right]_{\mathrm{W}} \left[\epsilon\right]_{\mathrm{PL}} \left[\epsilon\right]_{\mathrm{IB}} \left[\epsilon\right]_{\mathrm{OB}} \left[\epsilon\right]_{\mathrm{Ns}} \end{array} \right\rangle_{\mathrm{P}}$$

Second exchange

The second excerpt focuses on the asynchronous calls and the compensation measures. We assume that the agent checked the balance of the yellow account, and the balance is sufficient. The processors involved are the magenta agent, the yellow and the green account. The state of these three processors is:

$$\left\langle \begin{array}{c} \langle A_M \rangle_{\mathrm{PI}} \\ \left[\begin{array}{l} \left\langle \begin{array}{c} \langle 35 \rangle_{\mathrm{FI}} \left[\mathfrak{L}\right]_{\mathrm{T}} \left[\Omega; \upsilon\right]_{\mathrm{IL}} \\ \left[\epsilon\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{Is}} \left\{(A_Y, 38), (A_G, 40)\right\}_{\mathrm{As}} \end{array} \right\rangle_{\mathrm{F}}; \\ \langle\langle 3 \rangle_{\mathrm{FI}} \left[\mathfrak{L}\right]_{\mathrm{T}} \left[\Omega; \upsilon\right]_{\mathrm{IL}} \left[\epsilon\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{Is}} \langle \emptyset \rangle_{\mathrm{As}} \rangle_{\mathrm{F}}; \\ \langle\langle 2 \rangle_{\mathrm{FI}} \left[\mathfrak{S}\right]_{\mathrm{T}} \left[\Omega; \upsilon\right]_{\mathrm{IL}} \left[\epsilon\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{Is}} \langle \emptyset \rangle_{\mathrm{As}} \rangle_{\mathrm{F}} \end{array} \right]_{\mathrm{EL}} \\ \left[\epsilon\right]_{\mathrm{PL}} \left[\epsilon\right]_{\mathrm{IB}} \left[\epsilon\right]_{\mathrm{OB}} \left[\epsilon\right]_{\mathrm{Ns}} \end{array} \right\rangle_{\mathrm{P}}$$

$$\left\langle \begin{array}{c} \langle A_Y \rangle_{\mathrm{PI}} \\ \left[\langle\langle 38 \rangle_{\mathrm{FI}} \left[\mathfrak{S}\right]_{\mathrm{T}} \left[\epsilon\right]_{\mathrm{IL}} \left[A_M\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{As}} \langle \emptyset \rangle_{\mathrm{Is}} \rangle_{\mathrm{F}}\right]_{\mathrm{EL}} \\ \left[\epsilon\right]_{\mathrm{PL}} \left[\epsilon\right]_{\mathrm{IB}} \left[\epsilon\right]_{\mathrm{OB}} \left[\epsilon\right]_{\mathrm{Ns}} \end{array} \right\rangle_{\mathrm{P}}$$

$$\left\langle \begin{array}{c} \langle A_G \rangle_{\mathrm{PI}} \\ \left[\langle\langle 40 \rangle_{\mathrm{FI}} \left[\mathfrak{S}\right]_{\mathrm{T}} \left[\epsilon\right]_{\mathrm{IL}} \left[A_M\right]_{\mathrm{OL}} \left[\left(\left[\epsilon\right], A_M\right)\right]_{\mathrm{CL}} \langle \emptyset \rangle_{\mathrm{As}} \langle \emptyset \rangle_{\mathrm{Is}} \rangle_{\mathrm{F}}\right]_{\mathrm{EL}} \\ \left[\epsilon\right]_{\mathrm{PL}} \left[\epsilon\right]_{\mathrm{IB}} \left[\epsilon\right]_{\mathrm{OB}} \left[\epsilon\right]_{\mathrm{Ns}} \end{array} \right\rangle_{\mathrm{P}}$$

The agent produces the call with NEXT ACALL.

$$\frac{\begin{array}{c} A_Y \in \{A_Y, A_G\} \\ \emptyset \subseteq \{A_M, R_M, R_Y, R_G, A_Y, A_G\} \end{array}}{\begin{array}{c} \left[A_M, R_M, R_Y, R_G, A_Y, A_G\right]_{\mathrm{Ps}} \vdash \\ \left[\langle \left[\Omega \ldots\right]_{\mathrm{IL}} \langle \emptyset \rangle_{\mathrm{Is}} \left\{(A_Y, 38), (A_G, 40)\right\}_{\mathrm{As}} \rangle_{\mathrm{F}} \ldots\right]_{\mathrm{EL}} \\ \rightarrow \\ \left[\langle \left[\textsc{ACall}\ A_Y, \emptyset; \Omega \ldots\right]_{\mathrm{IL}} \langle \emptyset \rangle_{\mathrm{Is}} \left\{(A_Y, 38), (A_G, 40)\right\}_{\mathrm{As}} \rangle_{\mathrm{F}} \ldots\right]_{\mathrm{EL}} \end{array}} \textsc{next acall}$$

As with synchronous separate calls, a message is produced. However, no callback frame is provided for the supplier. Even though the call is asynchronous, the client still waits until the supplier enqueues the request.

$$\frac{\begin{array}{c}(A_Y, 38) \in \{(A_Y, 38), (A_G, 40)\} \\ \text{fresh}\,(44)\end{array}}{\begin{array}{c}\langle A_M \rangle_{\text{PI}} \vdash \\ [\langle [\text{ACALL}\ A_Y, \emptyset ...]_{\text{IL}}\ \langle \emptyset \rangle_{\text{IS}}\ \{(A_Y, 38), (A_G, 40)\}_{\text{AS}} \rangle_{\text{F}} ...]_{\text{EL}}\ [...]_{\text{OB}} \\ \rightarrow \\ [\langle [44 \downarrow A_Y ...]_{\text{IL}}\ \langle \emptyset \rangle_{\text{IS}}\ \{(A_Y, 38), (A_G, 40)\}_{\text{AS}} \rangle_{\text{F}} ...]_{\text{EL}} \\ [... \boxed{\text{ACall}\ A_M, 38, \emptyset}_{44} \uparrow A_Y]_{\text{OB}}\end{array}}\ \text{ACALL}$$

We skip the transmission to the yellow account processor and let it enqueue the request.

$$\frac{}{\begin{array}{c}\langle A_Y \rangle_{\text{PI}} \vdash \\ [... \langle \langle 38 \rangle_{\text{FI}}\ [...]_{\text{IL}}\ [A_M ...]_{\text{OL}} \rangle_{\text{F}} ...]_{\text{EL}} \\ [\boxed{\text{ACall}\ A_M, 38, \emptyset}_{44} ...]_{\text{IB}}\ [...]_{\text{OB}} \\ \rightarrow \\ [... \langle \langle 38 \rangle_{\text{FI}}\ [...\text{EXEC}\ \emptyset, \emptyset, \emptyset]_{\text{IL}}\ [A_M ...]_{\text{OL}} \rangle_{\text{F}} ...]_{\text{EL}} \\ [...]_{\text{IB}}\ [... \boxed{\text{OK}}_{44}]_{\text{OB}}\end{array}}\ \text{ENQ. ACALL}$$

After the transmission, the agent makes the request to the green account. This is very similar to the one before, so we do not show it here. The agent has enqueued the two requests. It does not need to wait for unlocking the two accounts, so it proceeds with NEXT UNLOCK.

$$\frac{}{[\langle [\Omega ...]_{\text{IL}} \rangle_{\text{F}} ...]_{\text{EL}} \rightarrow [\langle [\text{UNLOCK} ...]_{\text{IL}} \rangle_{\text{F}} ...]_{\text{EL}}}\ \text{NEXT UNLOCK}$$

To fulfill this instruction, two $\boxed{\text{Unlock}}$ messages are produced, which are transmitted and then handled with UNLOCK OK. For brevity, we only write the steps down once, for the yellow register, without transmission.

$$\frac{\begin{array}{c}\text{fresh}\,(46) \\ \langle A_M \rangle_{\text{PI}} \vdash \\ [\langle [\text{UNLOCK} ...]_{\text{IL}}\ \langle \{(A_Y, 38)\} \cup (A_G, 40) \rangle_{\text{AS}} \rangle_{\text{F}} ...]_{\text{EL}}\ [...]_{\text{OB}} \\ \rightarrow \\ [\langle [\text{UNLOCK} ...]_{\text{IL}}\ \langle (A_G, 40) \rangle_{\text{AS}} \rangle_{\text{F}} ...]_{\text{EL}}\ [... \boxed{\text{Unlock}\ A_Y, 38}_{46} \uparrow A_G]_{\text{OB}}\end{array}}{}\ \text{UNLOCK 1}$$

$$\frac{}{\begin{array}{c}[\langle \langle 38 \rangle_{\text{FI}}\ [...]_{\text{IL}}\ [A_Y ...]_{\text{OL}} \rangle_{\text{F}}]_{\text{EL}}\ [\boxed{\text{Unlock}\ A_Y, 38}_{46} ...]_{\text{IB}} \\ \rightarrow \\ [\langle \langle 38 \rangle_{\text{FI}}\ [...v]_{\text{IL}}\ [...]_{\text{OL}} \rangle_{\text{F}}]_{\text{EL}}\ [...]_{\text{IB}}\end{array}}\ \text{UNLOCK OK}$$

Then, UNLOCK FINISHED finishes the **separate** block.

$$\frac{\emptyset = \emptyset}{\begin{array}{c}[\langle [\mathfrak{L}]_{\text{T}}\ [\text{UNLOCK} ...]_{\text{IL}}\ \langle \emptyset \rangle_{\text{IS}} \langle \emptyset \rangle_{\text{AS}} \rangle_{\text{F}} ...]_{\text{EL}} \\ \rightarrow \\ [\langle [\mathfrak{L}]_{\text{T}}\ [...]_{\text{IL}}\ \langle \emptyset \rangle_{\text{IS}} \langle \emptyset \rangle_{\text{AS}} \rangle_{\text{F}} ...]_{\text{EL}}\end{array}}\ \text{UNLOCK FINISHED}$$

For the agent, the work is finished now. It can return from the call to `transfer`. However, the two accounts still need to process their requests. Even though the requests are to different features, they are essentially the same, so we only show the first one. The yellow register start execution of the feature, for which it does not need to acquire additional locks.

$$\frac{\emptyset \subseteq \emptyset}{\begin{array}{c}\left[\langle\left[\textsc{Exec}\ \emptyset, \emptyset, \emptyset \ldots\right]_{\textsc{Il}}\ \langle\emptyset\rangle_{\textsc{Is}}\langle\emptyset\rangle_{\textsc{As}}\rangle_{\textsc{F}\cdots}\right]_{\textsc{El}} \\ \rightarrow \\ \left[\langle\left[\Omega \ldots\right]_{\textsc{Il}}\ \langle\emptyset\rangle_{\textsc{Is}}\langle\emptyset\rangle_{\textsc{As}}\rangle_{\textsc{F}\cdots}\right]_{\textsc{El}}\end{array}}\ \textsc{exec}$$

It makes an uninteresting non-separate call, which we do not show here, and then registers a compensation for reverting the last change.

$$\frac{\emptyset \subseteq \emptyset}{\begin{array}{c}\left[A_M, R_M, R_Y, R_G, A_Y, A_G\right]_{\textsc{Ps}} \vdash \\ \left[\langle\left[\Omega \ldots\right]_{\textsc{Il}}\rangle_{\textsc{F}\cdots}\right]_{\textsc{El}} \\ \rightarrow \\ \left[\langle\left[\textsc{Register}\ \emptyset; \Omega \ldots\right]_{\textsc{Il}}\rangle_{\textsc{F}\cdots}\right]_{\textsc{El}}\end{array}}\ \textsc{next register}$$

$$\frac{}{\begin{array}{c}\left[\langle\left[\textsc{Register}\ \emptyset \ldots\right]_{\textsc{Il}}\ \left[\left(\left[\epsilon\right], A_M\right) \ldots\right]_{\textsc{Cl}}\rangle_{\textsc{F}\cdots}\right]_{\textsc{El}} \\ \rightarrow \\ \left[\langle\left[\ldots\right]_{\textsc{Il}}\ \left[\left(\left[\emptyset; \epsilon\right], A_M\right) \ldots\right]_{\textsc{Cl}}\rangle_{\textsc{F}\cdots}\right]_{\textsc{El}}\end{array}}\ \textsc{register}$$

It finishes the call with NEXT UNLOCK and UNLOCK FINISHED. We skip both and instead take a look at the following application of POP CLIENT 2.

$$\frac{}{\left[\langle\langle\mathfrak{S}\rangle_{\textsc{T}}\ \left[\upsilon \ldots\right]_{\textsc{Il}}\ \left[\left(\left[\emptyset\right], A_M\right)\right]_{\textsc{Cl}}\rangle_{\textsc{F}\cdots}\right]_{\textsc{El}} \rightarrow \left[\epsilon\right]_{\textsc{El}}}\ \textsc{pop client 2}$$

We can see that the compensation code, which was an **agent** that represents a call to `undo`, is discarded. This is because the frame was correctly closed by the client, the banking agent. After the other account also processed its instructions, the state of the processors is now:

$$\left\langle \begin{array}{c}\langle A_M\rangle_{\textsc{Pi}} \\ \left[\begin{array}{l}\langle\langle 3\rangle_{\textsc{Fi}}\ \left[\mathfrak{L}\right]_{\textsc{T}}\ \left[\Omega; \upsilon\right]_{\textsc{Il}}\ \left[\epsilon\right]_{\textsc{Ol}}\ \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}}\ \langle\emptyset\rangle_{\textsc{Is}}\langle\emptyset\rangle_{\textsc{As}}\rangle_{\textsc{F}};\right] \\ \langle\langle 2\rangle_{\textsc{Fi}}\ \left[\mathfrak{S}\right]_{\textsc{T}}\ \left[\Omega; \upsilon\right]_{\textsc{Il}}\ \left[\epsilon\right]_{\textsc{Ol}}\ \left[\left(\left[\epsilon\right], A_M\right)\right]_{\textsc{Cl}}\ \langle\emptyset\rangle_{\textsc{Is}}\langle\emptyset\rangle_{\textsc{As}}\rangle_{\textsc{F}}\right]_{\textsc{El}} \\ \left[\mathfrak{N}\right]_{\textsc{W}}\ \left[\epsilon\right]_{\textsc{Pl}}\ \left[\epsilon\right]_{\textsc{Ib}}\ \left[\epsilon\right]_{\textsc{Ob}}\ \left[\epsilon\right]_{\textsc{Ns}} \\ \langle\langle A_Y\rangle_{\textsc{Pi}}\ \left[\epsilon\right]_{\textsc{El}}\ \left[\mathfrak{N}\right]_{\textsc{W}}\ \left[\epsilon\right]_{\textsc{Pl}}\ \left[\epsilon\right]_{\textsc{Ib}}\ \left[\epsilon\right]_{\textsc{Ob}}\ \left[\epsilon\right]_{\textsc{Ns}}\rangle_{\textsc{P}} \\ \langle\langle A_G\rangle_{\textsc{Pi}}\ \left[\epsilon\right]_{\textsc{El}}\ \left[\mathfrak{N}\right]_{\textsc{W}}\ \left[\epsilon\right]_{\textsc{Pl}}\ \left[\epsilon\right]_{\textsc{Ib}}\ \left[\epsilon\right]_{\textsc{Ob}}\ \left[\epsilon\right]_{\textsc{Ns}}\rangle_{\textsc{P}}\end{array}\right\rangle_{\textsc{P}}$$

Aborted Transaction

What if the agent processor disappears after sending the `withdraw` command to the yellow account? There is never a υ enqueued, so at some point, the processor would be in the following state:

$$\left\langle \begin{matrix} \{R_M, R_Y, R_G, A_Y, A_G\}_{\text{PS}} \\ \langle A_Y \rangle_{\text{PI}} \left[\langle \langle 38 \rangle_{\text{FI}} \left[\mathfrak{S} \right]_{\text{T}} \left[\epsilon \right]_{\text{IL}} \left[A_M \right]_{\text{OL}} \left[(\left[\emptyset \right], A_M) \right]_{\text{CL}} \langle \emptyset \rangle_{\text{AS}} \langle \emptyset \rangle_{\text{IS}} \rangle_{\text{F}} \right]_{\text{EL}} \\ \left[\mathfrak{N} \right]_{\text{W}} \left[\epsilon \right]_{\text{PL}} \left[\epsilon \right]_{\text{IB}} \left[\epsilon \right]_{\text{OB}} \left[\epsilon \right]_{\text{NS}} \end{matrix} \right\rangle_{\text{P}}$$

With the agent no longer in the set of processors and no pending message in the in-box, the CLIENT MISSING rule can be applied.

$$\frac{}{\begin{matrix} \{R_M, R_Y, R_G, A_Y, A_G\}_{\text{PS}} \left[\epsilon \right]_{\text{IB}} \vdash \\ \left[\langle \left[\epsilon \right]_{\text{IL}} \left[(\left[\emptyset \right], A_M) \ldots \right]_{\text{CL}} \rangle_{\text{F}} \ldots \right]_{\text{EL}} \\ \rightarrow \\ \left[\langle \left[\text{REVERT}; \xi \right]_{\text{IL}} \left[(\left[\emptyset \right], A_M) \ldots \right]_{\text{CL}} \rangle_{\text{F}} \ldots \right]_{\text{EL}} \end{matrix}} \text{ CLIENT MISSING}$$

The REVERT instruction now starts to revert the changes. It takes the compensation registered last and executes it like a regular non-separate call.

$$\frac{}{\begin{matrix} \langle A_Y \rangle_{\text{PI}} \vdash \\ \left[\langle \left[\text{REVERT} \ldots \right]_{\text{IL}} \left[(\left[\emptyset; \epsilon \right], A_M) \ldots \right]_{\text{CL}} \rangle_{\text{F}} \ldots \right]_{\text{EL}} \\ \rightarrow \\ \left[\langle \left[\text{SCALL } A_Y, \emptyset; \text{REVERT} \ldots \right]_{\text{IL}} \left[(\left[\epsilon \right], A_M) \ldots \right]_{\text{CL}} \rangle_{\text{F}} \ldots \right]_{\text{EL}} \end{matrix}} \text{ REVERT}$$

We do not show the execution of the call here. If there were more compensations on the stack, they would now be executed too, one after another. Since there are none, the REVERT instruction is removed through REVERT FINISHED.

$$\frac{}{\begin{matrix} \langle A_Y \rangle_{\text{PI}} \vdash \\ \left[\langle \left[\text{REVERT} \ldots \right]_{\text{IL}} \left[(\left[\epsilon \right], A_M) \ldots \right]_{\text{CL}} \rangle_{\text{F}} \ldots \right]_{\text{EL}} \\ \rightarrow \\ \left[\langle \left[\upsilon \ldots \right]_{\text{IL}} \left[(\left[\epsilon \right], A_M) \ldots \right]_{\text{CL}} \rangle_{\text{F}} \ldots \right]_{\text{EL}} \end{matrix}} \text{ REVERT FINISHED}$$

The instruction made way for a υ, which now can remove the frame as if the call were successful.

6.12 Conclusion and Future Work

In chapter 5 we showed that the non-interference guarantees can be integrated into the language without an impact on efficiency compared to manual methods for synchronization. The semantics proposed for D-SCOOP shows it is possible to integrate the complex features of SCOOP, non-interference and lock-passing, with the concept of compensations in a model relying purely on message passing.

In the future, we want to look into using the semantics together with a model checker to be able to automatically reproduce message exchanges. Furthermore, we want to integrate the semantics in the existing executable frameworks for SCOOP [13, 47].

Chapter 7

Handling Parallelism in a Concurrency Model

Programming models for concurrency such as SCOOP are optimized for dealing with nondeterminism, for example to handle asynchronous events. To shield the developer from data race errors, such models may prevent shared access to data altogether. However, this restriction makes them unsuitable for applications that require data parallelism.

We present an API for permitting parallel access to arrays that internally circumvents the restrictions while preserving the safety guarantees of the original model. When applied to SCOOP, the approach exhibits a negligible performance overhead compared to ordinary threaded implementations of two parallel benchmark programs.

The content of this chapter is based on the paper *Handling Parallelism in a Concurrency Model* [60].

7.1 Introduction

Slicing is an API approach for parallel processing of shared-memory arrays as part of the framework of a concurrency model. To achieve this, the data structure is extended with features to obtain *slices*, i.e. contiguous data sections of the original data structure. These data parts can be safely used by multiple processors, and the race-freedom guarantee for the original data structure can be preserved.

The approach is applied to SCOOP. A performance evaluation using two benchmark programs (parallel Quicksort and matrix multiplication) shows that the approach is as fast as using threads, and naturally outperforms the original no-sharing approach. While SCOOP lends itself well to our approach, the basic idea can be helpful for providing similar extensions to active object-based models.

Since a distributed system such as D-SCOOP does not provide shared memory because of technical rather than safety reasons, the presented API does not offer any performance advantage. However, the API is designed in a way that it can fall back to an approach that uses copying, for example through the export mechanism suggested in chapter 5 or immutable objects in chapter 8. By doing so transparently, the programmer does not need to be concerned whether the processors are local or remote. For the rest of the chapter, we concentrate on the case where the processor have access to shared memory.

The technique focuses on arrays, but it shows an approach for controlled access to shared memory that can be applied to other divisible data structures, such as hash tables or trees. In comparison to immutable objects, it allows for sharing mutable data structures safely.

The remainder of the chapter is structured as follows. Section 7.2 describes the problem and the rationale of our approach. Section 7.3 presents the slicing technique. Section 7.4 provides the results of the performance evaluation. Section 7.5 describes related work and section 7.6 concludes with thoughts on future work.

7.2 Performance issues of race-free models

To help conquer the complexity of nondeterministic multithreading, programming models for concurrency may provide safety guarantees that eliminate common errors by construction. The SCOOP model, and therefore also D-SCOOP, does not allow sharing of memory between its regions that are handled by a single processor. More specifically, every object in SCOOP belongs to exactly one processor, its handler. Only the handler has direct access to the state of the object: all other processors have to communicate with the handler if they want to interact with the object. This prevents low-level data races, and the control mechanisms of SCOOP can be used to avoid high-level data races.

Unfortunately, the strict avoidance of shared memory has severe performance disadvantages when trying to parallelize various commonplace computational problems. As an example, listing 7.1 shows an in-place Quicksort algorithm written in SCOOP. Every time the array is split, a new worker is created to sort its part of the array. The workers `s1` and `s2` and the array `data` are denoted as **`separate`**, that is, they are references to objects that may have a different handler. By creating a separate object, a new processor is spawned. Each processor can only be controlled by one other processor at a time, thereby ensuring freedom from data races.

The execution of this example exhibits parallel slowdown: a sequential version outperforms the algorithms for most workloads. This has two main reasons:

1. Every call to the data array involves adding the call to the private queue, removing the call from the private queue, and sending back the result; this creates a large communication overhead.
2. Only one of the workers at a time can execute the `get` and `swap` features on the array because they require control of the processor handling the array; this serialization prevents the algorithm from scaling up.

The same issues occur in a broad range of data-parallel algorithms using arrays. Efficient implementations of such algorithms are impossible in active object concurrency models such as SCOOP. Any viable solution to the problem has to get rid of the communication overhead and the serialization. There are three general approaches to this problem:

1. Weaken the concurrency model to allow shared memory without race protection, or interface with a shared memory language. The programmers are responsible to take appropriate synchronization measures themselves.
2. Use immutable data structures and work on copies.
3. Enable shared memory computations, but hide it in an API that preserves the race-freedom guarantees of the concurrency model.

The first approach trades the safety guarantees of the model for unrestricted access to shared memory, partially undermining the effort of using the model in the first place. Still, it is the most common approach. The second approach is safe, but may require different algorithms and requires more memory. If this is a viable alternative, the *immutable classes* presented in chapter 8 can fill the gap. In case it is not viable, we present here a solution based on the third approach, in particular offering both race protection and shared memory performance.

7.3 Array slicing

To allow the implementation of efficient parallel algorithms on arrays, two types of array manipulation have to be supported:

- *Parallel disjoint access:* each processor has read and write access to disjoint parts of an array.
- *Parallel read:* multiple processors have read-only access to the same array.

The array slicing technique presented in this section enables such array manipulations by defining two data structures, *slices* and *views*, representing parts of an array that can be accessed in parallel while maintaining race-freedom guarantees.

Slice Part of an array that supports read and write access of single threads.

View Proxy that provides read-only access to a slice while preventing modifications to it.

In the following, we give a specification of the operations on slices and views.

7.3.1 Slices

Slices enable parallel usage patterns of arrays where each thread works on a disjoint part of the array. The main operations on slices are:

Slicing Creating a slice of an array. It transfers some of the data of the array into the slice. If shared memory is used, the transfer can be done efficiently using aliasing of the memory and adjusting the bounds of the original array.

Merging The reverse operation of slicing. Merging two slices requires them to be adjacent to form an undivided range of indexes. The content of the two adjacent slices is transferred to the new slice, using aliasing if the two are also adjacent in shared memory. Merging ore than two slices can be done in multiple steps that merge two slices each[1].

Based on this central idea, an API for slices can be defined as in listing 7.2. Note that this is a simplified API; the actual API uses inheritance to give a common interface to both slices and views. The letter `T` refers to the type of the array elements.

After creating a new slice using `make`, the slice can be used like a regular array using `item` and `put` with the `indexes` ranging from `lower` to `upper`, although modifying it is only allowed if `is_modifiable` is true, which is exactly if `readers` is zero.

Internally, the attribute `area` is a direct pointer into memory which can be accessed like a 0-based array. The `base` represents the base of the slice, which is usually 1 for Eiffel programs, but may differ when a merge results in a copy. The operations `freeze` and `melt` increment and decrement the `readers` attribute, which influences `is_modifiable`, and are used by views (see section 7.3.2).

[1] In principle, the API could also provide a merge feature that accepts an arbitrary number of slices

Slicing. Like any other object, a reference to the slice can be passed to other processors. A processor having a reference to a slice can decide to create a new slice by slicing from the lower end (`slice_head`) or upper end (`slice_tail`). By doing this, the original slice transfers data to the new slice by altering the bounds and referencing the same memory if possible. Freedom from race conditions is ensured through exclusive access to the disjoint parts.

Listing 7.3 shows an implementation of the `slice_head` creation procedure, taking advantage of shared memory. It copies the `lower` bound, the `base` and the memory reference of the slice `a_original`. It also sets the upper bound according to the size `n` of the new slice and increases the lower bound of the original slice by `n`.

Merging. If a processor has two *adjacent* slices (the lower index of one slice equals the upper index of the other plus one), calling `merge` creates a new combined slice. This transfers all the data from the old slices to the new one, making the old ones empty. If the two slices are located next to each other in memory, the transfer simply adjusts the bounds; otherwise, it copies the data into a new slice.

The implementation of merging (see listing 7.4) sets the bounds according to the arguments. It then checks whether the two parts are actually next to each other in memory by checking whether the `area` and the `base` are the same. In this case, it copies the base and the memory reference. Otherwise, it allocates new memory and copies all the data of the two arguments. In the end, it empties the two arguments, setting their `count` to 0 by making `lower` = 1 and `upper` = 0.

Strategies for slicing. The most common choice for disjoint index subsets are sets with contiguous indexes. Those subsets can be identified by their lower and upper index and resemble a normal array. A rarer case is to create the disjoint subsets according to another principle. This warrants a different implementation, which is possible by using inheritance. However, current cache architectures limit the usefulness of chunks with a size smaller than a cache line.

7.3.2 Views

Views enable read-only access to arrays. The difference between views and immutable classes is that a view is a thin layer over a slice, so it shares the same data. The main operations on views are defined as follows:

Viewing Creating a view from a slice copies the bounds and the memory reference into the view. The original slice is no longer modifiable.

Releasing The reverse operation of viewing. If no other views on the same slice exist, it is modifiable again. Also, the view is no longer usable.

The API for views is shown in listing 7.5. A processor is able to read the slice in parallel by creating a view using the `make` creation procedure. The original slice is then available as the `original` query. This also prevents all further modifications of the array unless the view is released with the `free` procedure. All the other features of views behave exactly like their counterparts in the slices.

Viewing. Creating a view basically copies the bounds and the memory reference into the view. By increasing the view counter (`readers`) using the `freeze` operation of the slice `a_original`, the original slice is no longer modifiable (see listing 7.6). By

		Number of cores					
		1	2	4	8	16	32
Quicksort	Slicing	157.4	147.1	81.9	66.4	59.9	59.2
	Threads	158.6	145.1	82.8	68.0	61.5	59.8
Matrix multiplication	Slicing	184.8	95.0	51.2	24.0	14.1	7.3
	Threads	178.0	91.7	46.6	23.6	12.6	7.3

Table 7.1: Mean running times (in seconds)

calling `free` on a view, the view loses its reference to the memory of the slice and the original slice is notified through `melt` that there is one less `reader`.

Releasing. The `free` procedure redirects the `area` to 0 and sets `lower` to 1 and `upper` to 0. Therefore no access is possible at any index. In addition, the number of readers of the original is decremented by a call to `melt`. This causes the original to be modifiable again if the number of readers falls to zero. Because of its simplicity we chose not to show the code here. Note that if the garbage collector collects a view, the view is also released. Since garbage collection is, unless manually started, non-deterministic, relying on this mechanism is not recommended.

7.4 Performance evaluation

To assess the performance of our approach, we apply it to two benchmark problems: to determine how well our approach works in a divide-and-conquer scenario, we choose a parallel in-place Quicksort algorithm; to determine the raw performance, we use parallel matrix multiplication. In both cases, the extension of SCOOP with the slicing technique is compared with implementations in Eiffel using only threads and without synchronization except a join at the end.

For the performance tests we used a server with four 8-core Intel Xeon E7-4830 processors and 256 GB of RAM. We ran every program 20 times and report the mean value of the running times in table 7.1. The source code of the benchmarks is available online[2] as well as the support for slicing in SCOOP[3]. In the following we discuss both benchmarks and their results in detail.

7.4.1 Quicksort

Listing 7.7 shows the Quicksort example implemented using slices instead of regular arrays (compare section 7.2). We omitted the actual sorting code to focus on slicing and merging. The main difference is the usage of `slice_head` and `slice_tail` instead of storing the bounds in variables. The implementation of `sort` can stay the same, although there is no need for storing the bounds and extra features for swapping and retrieving since data is no longer separate.

For the performance measurement, the Quicksort benchmark sorts an array of size 10^8, which is first filled using a random number generator with a fixed seed.

[2] `https://bitbucket.org/mischaelschill/array-benchmarks`
[3] `https://bitbucket.org/mischaelschill/scoop-library`

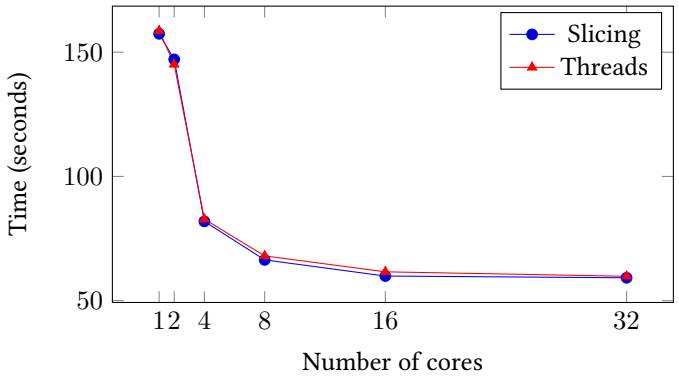


Figure 7.1: Quicksort: scalability

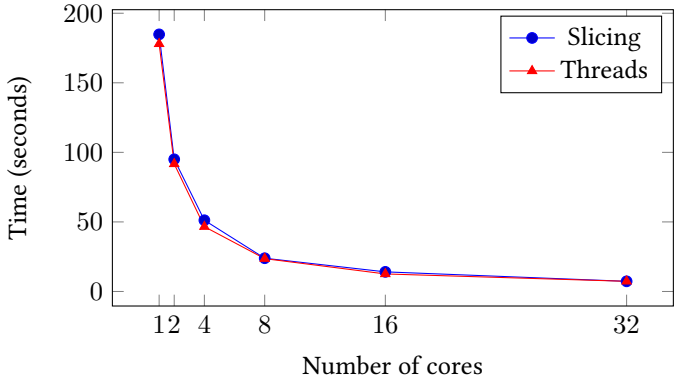


Figure 7.2: Matrix multiplication: scalability

The benchmarked code is similar to listing 7.7, but also adds a limit on the number of processors used. As evident from fig. 7.1, the performance characteristics of the slicing technique and threading is almost identical.

7.4.2 Matrix multiplication

Listing 7.8 shows a class facilitating parallel multiplication of matrices, using a two dimensional version of slices and views (**`SLICE2`** and **`SLICE_VIEW2`**, implemented in a very similar fashion to the one-dimensional version discussed in section 7.3.1). The worker is created using `make` which slices off the first `n` rows into `product`. The `multiply` command fills the slice with the result of the multiplication of the left and right matrices. Afterwards, the views are decoupled using `free`. Dividing the work between multiple workers and merging the result is left to the client of the worker.

For the performance measurement, the matrix multiplication test multiplies a 2000 to 800 matrix with an 800 to 2000 matrix. Figure 7.2 shows again similar performance characteristics between slicing and threads.

7.5 Related work

We are not aware of any programming model supporting slicing while avoiding race conditions. However, similar means to create an alias to a subset of an array's content are common in most programming languages or their standard library. For example, the standard library of Eiffel [21] can create subarrays. Perl [55] has language support for slicing arrays. Slices and slicing are a central feature of the Go programming language [23]. However, these slicing solutions were not created with the intention of guaranteeing safe access: the portion of memory aliased by the new array/slice remains accessible through the original array, which can lead to race conditions if two threads access them at the same time.

Enabling many processors to access different parts of a single array is a cornerstone of data parallel programming models. OpenMP [16] is the de-facto standard for shared-memory multiprocessing. Its API offers various data parallel directives for handling the access to arrays, e.g. in conjunction with parallel-for loops. Threading Building Blocks [57] is a C++ library which offers a wide variety of algorithmic skeletons for parallel programming patterns with array manipulations. Chapel [12] is a parallel programming language for high-performance computation offering concise abstractions for parallel programming patterns. Data Parallel Haskell [11] implements the model of nested data parallelism (inspired by NESL [8]), extending parallel access also to user-defined data structures. In contrast to our work, these approaches focus on efficient computation but not on safety guarantees for concurrent access, which is our starting point.

The concept of views is an application of readers-writers locks, first introduced by Courtois, Heymans and Parnas [14], tailored to the concept of slices.

The introduction of *passive regions* into Scoop by Morandi et al. [46], as well as optimizations by West [66], reduce the communication overhead of shared data structures, but does not permit parallel access.

7.6 Conclusion

While programming models for concurrency and parallelism have different goals, they can benefit from each other: concurrency models provide safety mechanisms that can be advantageous for parallelism as well; parallelism models provide performance optimizations that can also be profitable in concurrent programming. We have taken a step in this direction by extending Scoop with a technique for efficient parallel access of arrays, without compromising the original data-race freedom guarantees of the model. An important insight from this work is that safety and performance do not necessarily have to be trade-offs: results on two typical benchmark problems show that our approach has the same performance characteristics as ordinary threading.

In future work, we want to investigate whether views can be marked as immutable so that they can be passed to other processors without creating another view. Another point of interest is a more direct integration of slices into the D-Scoop runtime to improve the performance especially when slices are used together with distribution.

```
data: separate ARRAY[T]
lower, upper: INTEGER

make (d: separate ARRAY[T]; n: INTEGER)
    do
        if n > 0 then
            lower := d.lower
            upper := d.lower + n - 1
        else
            upper := d.upper
            lower := d.upper + n + 1
        end
        data := d
    end

sort
    local
        tmp, i, j: INTEGER
        s1, s2: separate SORTER[T]
    do
        if upper > lower then
            pivot := get (data, upper)
            from i := lower; j := lower    until i = upper loop
                separate data as c_data do
                    if c_data[i] < pivot then
                        tmp := c_data[i];
                        c_data[i] := c_data[j];
                        c_data[j] := tmp;
                        j := j + 1
                    end
                i := i + 1
            end
            separate data as c_data do
                tmp := c_data[upper];
                c_data[upper] := c_data[j];
                c_data[j] := tmp;
            end
            create s1.make (data, j - lower)
            create s2.make (data, j - upper)
            separate_sort(s1, s2)
        end
    end

separate_sort (s1, s2: separate SORTER[T])
    do
        s1.sort
        s2.sort
    end
```

Listing 7.1: SORTER: In-place Quicksort in Scoop

```
frozen class
    SLICE[T]

create -- Creation procedures (constructors)
    make, slice_head, slice_tail, merge

feature {NONE}
    make(n: INTEGER)
        -- Create a new slice with a capacity of ‘n’
    slice_head(n: INTEGER; slice: separate SLICE)
        -- Slice off the first ‘n’ entries of ‘slice’
    slice_tail(n: INTEGER; slice: separate SLICE)
        -- Slice off the last ‘n’ entries of ‘slice’
    merge(a, b: separate SLICE)
        -- Create a new slice by merging ‘a’ and ‘b’

feature -- Queries
    item(index: INTEGER): T
        -- Retrieve the item at ‘index’
    indexes: SET[INTEGER]
        -- Indexes of this slice
    lower: INTEGER
        -- Lowest index of the index set
    upper: INTEGER
        -- Highest index of the index set
    count: INTEGER
        -- Number of indexes: ‘upper - lower + 1’
    is_modifiable: BOOLEAN
        -- Whether the array is currently modifiable, i.e.\ ‘readers = 0’
    readers: INTEGER
        -- The number of views on the slice

feature -- Commands
    put(value: T; index: INTEGER): T
        -- Store ‘value’ at ‘index’
    freeze
        -- Notifies the slice that a view on it is created by incrementing ‘readers’
    melt
        -- Notifies the slice that a view on it is released by decrementing ‘readers’

feature {NONE} -- Internals
    area: POINTER
        -- Direct unprotected memory access
    base: INTEGER
        -- The offset into memory
```

Listing 7.2: API for slices

```
slice_head (a_original: separate SLICE[T]; n: INTEGER)
    require -- Precondition
        within_bounds: n > 0 and n <= a_original.count
        a_original.is_modifiable
    do
        lower := a_original.lower
        upper := a_original.lower + n - 1
        base := a_original.base
        area := a_original.area
        a_original.lower := a_original.lower + n
    ensure -- Postcondition
        a_original.count = old a_original.count - n
        a_original.lower = old a_original.lower + n
        a_original.upper = old a_original.upper
        lower = old a_original.lower
        upper = a_original.lower - 1
        count = n
        -- "forall i in indexes : item(i) = old a_original.item(i)"
    end
```

Listing 7.3: Slicing

```
merge (a_one, a_another: separate SLICE[T])
    require
        a_one.is_modifiable
        a_another.is_modifiable
        one.is_adjacent (a_another)
    do
        lower := a_one.lower.min(a_another.lower)
        upper := a_another.upper.max(a_one.upper)
        if  a_one.area = a_another.area and a_one.base = a_another.base then
            area := a_one.area
            base := a_one.base
        else
            base := lower
            -- "Copy data from the a_one and a_another to area"
        end
        a_another.empty; a_one.empty
    ensure
        lower = old a_one.lower.min(a_another.lower)
        upper = old a_one.upper.max(a_another.upper)
        a_one.count = 0 and    a_another.count = 0
        -- "forall i in old a_one.indexes : item(i) = old a_one.item(i)"
        -- "forall i in old a_another.indexes : item(i) = old a_another.item(i)"
    end
```

Listing 7.4: Merging

```
class
    VIEW[T]

create
    make

feature {NONE} -- Creation procedures (constructors)
    make(slice: separate SLICE[T])
        -- Create a new view on ‘slice’

feature -- Queries
    original: separate SLICE[T]
        -- Slice this view references
    indexes: SET[INTEGER]
        -- Indexes of this view
    item(index: INTEGER): T
        -- Retrieve the item at index
    lower: INTEGER
        -- Lowest index of the index set
    upper: INTEGER
        -- Highest index of the index set

feature -- Commands
    free
        -- Disconnects the view from the slice

feature {NONE} -- Internals
    area: POINTER
        -- Direct unprotected memory access
    base: INTEGER
        -- Offset into memory
```

Listing 7.5: API for slice views

```
make (a_original: separate SLICE[T])
    do
        a_original.freeze
        original := a_original
        lower := a_original.lower
        upper := a_original.upper
        base := a_original.base
        area := a_original.area
    ensure
        lower = a_original.lower
        upper = a_original.upper
        not a_original.is_modifiable
        -- "forall i in indexes : item(i) = a_original.item(i)"
    end
```

Listing 7.6: Viewing

```
class SLICE_SORTER[G -> COMPARABLE]

create
    make, slice_top, slice_bottom

feature {NONE}
    make (a_data: SLICE[G]; a_workers: INTEGER)
        -- Sorts ‘a_data’ using ‘a_workers’ workers
        do workers := a_workers; data := a_data end

    make_head (a_data: separate SLICE[G]; n, a_workers: INTEGER)
        do workers := a_workers; create data.slice_head (n, a_data) end

    make_tail (a_data: separate SLICE[G]; n, a_workers: INTEGER)
            -- Similar to slice_top

feature
    workers: INTEGER
    data: SLICE[G]
    sort
        local
            middle: INTEGER
            separate_sorter1, separate_sorter2: separate SLICE_SORTER[G]
        do
            if data.count > 1 then
                middle := data.count // 2 + data.lower
                -- Quicksort partitioning and swaps
                ...
                if workers > 1 then
                    create separate_sorter1.make_head
                        (data, middle - data.lower, workers // 2)
                    create separate_sorter2.make_tail
                        (data, data.upper - middle, workers // 2)
                    separate
                        separate_sorter1 as c_separate_sorter1,
                        separate_sorter2 as c_separate_sorter2
                    do
                        separate_sorter1.sort
                        separate_sorter2.sort
                        create data.merge (data, separate_sorter1.data)
                        create data.merge (data, separate_sorter2.data)
                    end
                else -- Sort sequentially, code omitted
                end
            end
        end
end
```

Listing 7.7: Quicksort algorithm using slices

```
left, right: SLICE_VIEW2[T]
product: SLICE2[T]

make (l, r, p: separate SLICE2[T]; n: INTEGER)
    do
        create left.make(l); create right.make(r)
        create product.slice_top (n, p)
    end

multiply
    local
        k, i, j: INTEGER
    do
        from i := product.first_row until i > product.last_row loop
            from j := product.first_column until j > product.last_column loop
                from k := left.first_column until k > left.last_column loop
                    product[i, j] := product[i, j] + left[i, k] * right [k, j]
                    k := k + 1
                end
                j := j + 1
            end
            i := i + 1
        end
        left.free; right.free
    end
```

Listing 7.8: Matrix multiplication worker using slices and views

Chapter 8

Immutable Classes

Immutable objects never change. Immutability is a useful property: it helps reasoning and enables optimizations. If, for example, a string object is immutable, then a supplier that receives a reference to a string can keep it without risking a change to the string afterwards. With mutable strings, the supplier needs to make a copy of the string to make sure it cannot be changed. Since oftentimes the mutable string would not be changed afterwards, these copies are often superfluous.

In the context of SCOOP, immutable objects can always be treated as non-separate: without the ability to change their state, no data races are possible. This is especially useful for large objects like strings or matrices, where duplication through import or export requires significant processor time and memory. With the expansion of SCOOP to D-SCOOP, immutable objects can be transparently copied over to other nodes without impact on the semantics.

Furthermore, in cases of restricted memory capacities, immutable objects with the same state can be merged; a prime candidate for this are strings.

Our formal approach is based on judgments over the program text. This is equal to the static part of an operational semantics [56]. Whereas for most operational semantics the static part is mainly concerned with type checking and type checking is, in comparison with immutability checking, simple, our approach needs significantly more judgments.

The chapter starts with a motivating example in section 8.1 before it goes into the technical details. These start with a description of the abstract syntax tree that is used for structural induction in section 8.2. Section 8.3 then continues with some general judgments and rules that are used later. Section 8.4 introduces conformance as used by this framework, which is a simplified version of the full SCOOP conformance rules, but it is sufficient to show immutability. The multiple inheritance scheme employed by Eiffel requires a complex logic to resolve features and their actual context. Since it is important for our approach to resolve features, we describe the functions and judgments related to resolving features in section 8.5. Sections 8.6 to 8.10 describe the high level properties together with the associated judgments and inference rules. At last, section 8.12 concludes with a comparison to other approaches.

8.1 Motivation

We use a compiler as a motivating example, more specifically, the abstract syntax tree constructed by the parser and used throughout the process. The tasks in our hypothetical compiler are the following:

1. The input is parsed to create abstract syntax trees.
2. Creation of various indexes for classes, features etc.
3. Type checking takes place.
4. Other static analyses are run.
5. Code is optimized.

Although these tasks are dependent upon previous tasks to be, at least partially, completed, the tasks themselves offer many opportunities for parallelization. Since computers used by developers usually have multiple cores, this is an important aspect.

The abstract syntax trees and the indexes are large and complex data structures used by most tasks. A simple solution is to put the indexes and the AST's on different processors, however, this basically serializes all access to the data structures. With transparent optimizations, this serialization, on a shared memory machine, the serialization can be done using simple locks. However, it still represents a bottleneck on systems with a large number of cores, especially since accesses are very frequent.

We can observe that it is possible to restrict access to the index to reads only: The output of the various tasks does not need to be put into the data structures of the input. From a software engineering perspective, this has the added advantage of keeping the classes of the abstract syntax tree small.

With this observation, we can argue that we can make copies of the whole structure and provide the worker processors with one copy each. Since we do not necessarily need significantly more workers than cores in the machine, this is a reasonable solution. Once the copies are created, the bottleneck is removed. However, this requires a significant increase in memory usage and, at the same time, adds some computational overhead and complexity. The latter mainly stems from the fact that it is not possible to create expanded classes for this purpose since expanded variables are not polymorphic and there is no expanded version of an array[1].

If look at this problem from a distance, we can see that we only copied the data because of the Scoop rules. Since we do not modify the shared data structures, there is no risk of data races and the Scoop rules are overbearing. We could introduce a built-in mechanism similar to slicing as discussed in chapter 7 for more data structures. Another possible solution, the one this chapter is dedicated to, is to have a mechanism for statically proving data structures, or more generally object, to be immutable and then adding an exception to the Scoop rules that allows parallel access to these objects. The same way that expanded objects, due to their semantics, are exempt from the Scoop typing rules, immutable objects can be exempt.

The gentle reader might ask what immutable objects have to contribute to D-Scoop. While any improvements to Scoop by extension also benefit D-Scoop, the possibilities for optimizations with immutable objects also directly affect D-Scoop. Immutability not only allow safe parallel access, it also allows for copies to be indistinguishable from the original. It is therefore possible to transparently copy an immutable object that is an argument or result of a remote call to the recipient. It is

[1] More specifically, the `SPECIAL` class

often more efficient to copy a large data structure if it means that a few remote calls are thereby avoided.

In the context of our example, computationally expensive optimizations could be delegated to remote workers. In fact, distributed compilation is used in practice, the distcc[2] program can be used for distributed C compilation. While the export mechanism suggested in chapter 5 can achieve the same effect, immutable objects can make this transparent. Furthermore, if all copies are identified correctly, redundant copies can be avoided if the same object gets passed back and forth between nodes.

8.2 Abstract Syntax Tree

To introduce the framework, we use a simplified version of Eiffel but without omitting constructs that affect the validity of our reasoning. In particular, we retain the full multiple inheritance system of Eiffel since it is one of the more difficult parts for reasoning and it affects immutability.

Table 8.1 shows the syntax we use. Every node has a (possibly empty) list of children, which are ordered in the same way as in the concrete syntax. We use modifiers ?, * and | as they are used in regular expressions, but instead of using parentheses, we underline the scope of the modifier if it does not apply just to the symbol immediately preceding it. Figures 8.1 and 8.2 give a graphical overview of the abstract syntax tree.

Some node types, such as *body*, are virtual nodes, which is indicated through the usage of an italic font and a grey background the graphical tree. They are a placeholder for other nodes; in the case of *body*, this would be *do*, *once*, *attribute* or *deferred*. This means that in the actual abstract syntax tree of a program, in places where there is a virtual node in the syntax description, one of these actual node is placed. In other words, the children of a virtual node replace their parent in the actual abstract syntax tree.

We assume correct programs under the rules of Eiffel as in [41].

8.2.1 Nodes and Lists

As mentioned above, every node has a list of children. We use the symbol ϵ to indicate an empty list, and lists are formed according to the following principle, with l being a list and i an item:

$$l := il|\epsilon$$

The first item of a list corresponds to the first child node occurring in the program text, the second item to the second and so on.

As an example, we look at a short instruction list with an assignment and a command that corresponds to the following excerpt:

```
a := 15;
f (a);
```

This is equivalent to this abstract syntax tree:
assign[name[a] int[15]] uc[name[f] name[a]]

[2] https://github.com/distcc

AST Node	Concrete Form	Description
prog	cls*	Program
cls	cflag name [param ; param*]? inh* feat* inv* **end**	Class
cflag	**deferred class** | **expanded class** | **class**	Class Flag
param	name <- type?	Parameter
inh	**inherit** type rename* export* undef* redef* select* **end**?	Inheritance
type	name [type , type*]?	Type
rename	**rename** name **as** name	Rename clause
export	**export** {name} name	Export clause
undef	**undefine** name	Undefine clause
redef	**redefine** name	Redefine clause
select	**select** name	Select clause
feat	**feature** visib name args : **separate**? type? req? local? *body*? ens?	Feature
visib	{ name } ?	Visibility
args	(var ; var*)?	Arguments
var	name : **separate**? type	Variable
req	**require** **else**? *exp**	Precondition
local	**local** var*	Locals
body	do | once | attribute | deferred	Feature Body
do	**do** *instr** rescue?)	Routine body
deferred	**deferred**	Deferred body
once	**once** oscope? *instr** rescue?	Once routine body
attribute	**attribute** *instr** rescue?	Attribute initializer
oscope	("PROCESS" | "THREAD" | "OBJECT")	Once scope
rescue	**rescue** *instr**	Rescue clause
ens	**ensure** **then**? *exp**	Postcondition
inv	**invariant** *exp*	Invariant
instr	qc | uc | assign | createi | cond | loop	Instruction
qc	*exp*.name (*exp* , *exp**)?	Qualified call
uc	name (exp , exp*)?	Unqualified call
assign	name := *exp*	Assignment
createi	**create** name . name (*exp* , *exp**)?	Create instruction
cond	**if** *exp* **then** *instr** elseif* else? **end**	Conditional
elseif	**elseif** *exp* **then** *instr**	Else-if branch
else	**else** *instr**	Else branch
loop	**until** *exp* vari? inv? **loop** instr* **end**	Loop
vari	**variant** *exp*	Loop variant
exp	qc | uc | createe | char | str | int | real | bool	Expression
createe	**create** { name }. name (*exp* , *exp**)?	Create expression

Table 8.1: Simplified Eiffel Abstract Syntax Tree

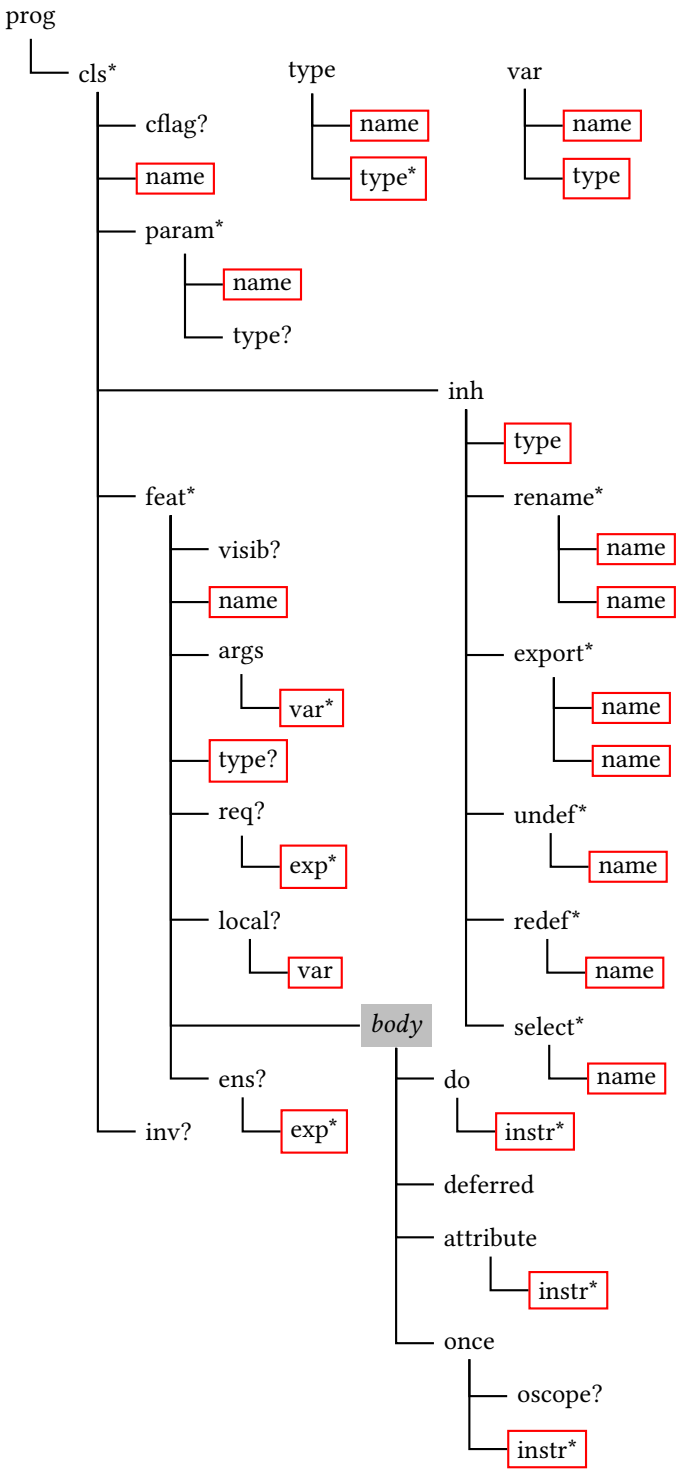


Figure 8.1: Eiffel syntax: classes and features

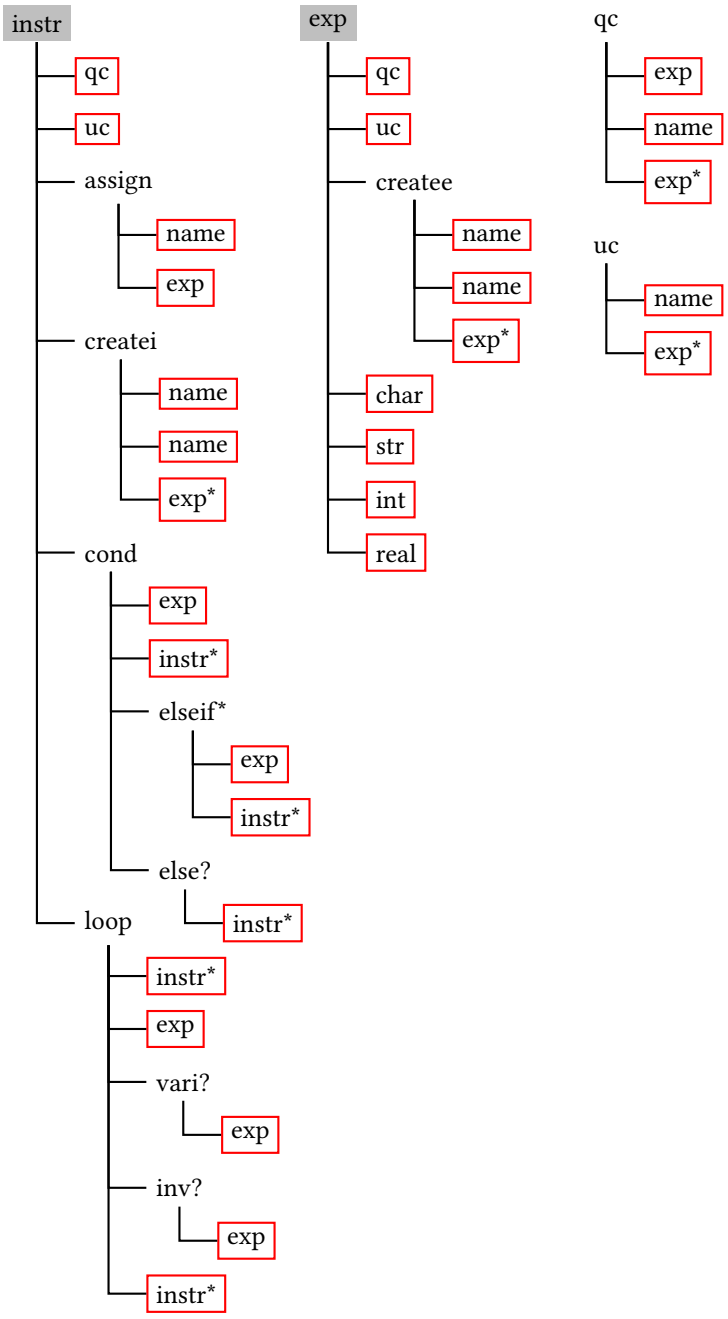


Figure 8.2: Eiffel syntax: instructions and expressions

Variable typing

Variables can substitute AST nodes and lists of AST nodes. We assert the admissible content of a variable with the following judgment, where v is a variable and t is an AST node type, with $n \in \mathbb{N}$:

$$v \wr t_0, t_1, ..., t_n \tag{8.1}$$

Note that v above cannot be a list, although variables may contain lists of nodes. To allow v to be a list of the given types as well as ϵ, we use the following notation:

$$v \wr^* t_0, t_1, ..., t_n \tag{8.2}$$

We also introduce a short-hand notation for single AST node types by adding them as a superscript to variables in the conclusion:

v^t as equivalent to $\dfrac{v \wr t}{v}$

In the same fashion, we define the short-hand notation

v^{t*} for $\dfrac{v \wr^* t}{v}$

Filtering

We introduce a filter operator $\triangledown$ that extracts only nodes of specific types from a list.

$$(i\ l)\triangledown j = i\ (l \triangledown j) \iff i \wr l \tag{8.3}$$
$$(i\ l)\triangledown j = l \triangledown j \iff i \not\wr l \tag{8.4}$$
$$\epsilon \triangledown j = \epsilon \tag{8.5}$$

List content

We use the symbol $\in$ to assert that the node on the left appears in a list on the right. This is defined as follows, with i, j being nodes and l a list.

$$i \in (j\ l) \iff i = j \vee i \in l \tag{8.6}$$

We also define $\notin$ for asserting that the item on the left is not in the list on the right.

$$i \notin (j\ l) \iff i \neq j \wedge i \notin l \tag{8.7}$$

Furthermore, we define the function $\mathfrak{H}$ for extracting the head of a list, or an empty list if the list is empty.

$$\mathfrak{H}(t...) = t \tag{8.8}$$
$$\mathfrak{H}(\epsilon) = \epsilon \tag{8.9}$$

Existential quantification

We often need to existentially quantify a node or a list that is present but not referred to again. We use the question mark symbol (?) as a placeholder for a single node and an ellipsis (...) for a list which may also be empty.

List concatenation

We use the symbol $+$ to denote list concatenation. For its definition, we use a helper operator $\bar{+}$.

$$a + b = (a \bar{+} \epsilon) \bar{+} b \tag{8.10}$$
$$(i, l) \bar{+} b = l \bar{+} (i, b) \tag{8.11}$$
$$\epsilon \bar{+} b = b \tag{8.12}$$

Basic nodes and naming

We assume the following basic nodes for simple values:

`int` Integer constant
`real` Floating point constant
`char` Character constant
`str` String constant

Alphanumeric identifiers in the program text are referred to as names. Nodes for names are written as `name[`t`]` with t being the name written in the text.

For example, the program text `a := 15` is equivalent to assign[name[a] int[15]].

For conciseness, we use the special class names **`ANY`**, **`NONE`** and **`IMMUTABLE`** without the surrounding node, so these are equivalent to name[ANY], name[NONE] and name[IMMUTABLE]. In the same fashion, **`Result`** stands for name[Result].

8.3 General Statements

Due to the richness of Eiffel, we require significantly more judgments to show that a class is immutable than for simple type checks. Also, we are formally defining every judgment, whereas some literature may omit what they seem obvious. To simplify understanding and reading, we use natural language instead of symbols for the judgments. Since this leads to longer judgments, we write the premises atop instead of next to each other if the space does not permit the latter.

Before we introduce the judgments, rules and properties related to immutability, we need to establish some more general judgments and functions.

8.3.1 Global Context

All the described judgments contain the local context in which they are true, given that the program does not change. This allows us to add every derivable judgment to the global context by using the following structural rule, which allows for deriving a judgment early:

$$\frac{\Gamma \vdash \Delta \qquad \Gamma, \Delta \vdash \Theta}{\Gamma \vdash \Theta}$$

8.3.2 Existence

The program and all its classes *exist*. This is the general context of the derivation and represented by the judgment

$$c \text{ exists.} \tag{8.13}$$

where c is a class or a program, that is, a list of classes. From the existence of a list of classes, we can derive the existence of every class in the list:

$$\frac{\Gamma \vdash p^{\text{prog}} \text{ exists.} \qquad \Gamma \vdash c \in p}{\Gamma^3 \vdash c^{\text{class}} \text{ exists.}}$$

8.3.3 Defined features

Similar to the existence of classes, we can assert the existence of a feature with a specific name in a given class. The judgment

$$f^{\text{feat}} \text{ is feature } n^{\text{name}} \text{ of } c^{\text{name}}. \tag{8.14}$$

states that the feature node f is defined in class c as n. Resolving features given a specific context is a different matter discussed in section 8.5. This judgment can be derived from the existence of the class:

$$\frac{\Gamma \vdash \text{cls} \left[\ ?, n, c\ \right] \text{ exists.} \qquad \text{feat} \left[\ v, f, b\ \right] \in c}{\Gamma \vdash \text{feat} \left[\ v, f, b\ \right] \text{ is feature } f^{\text{name}} \text{ of } n^{\text{name}}.}$$

8.3.4 Features of a Class

The judgment

$$n^{\text{name}^*} \text{ features of } A^{\text{name}}. \tag{8.15}$$

declares that the set of names n is the defined and inherited features of class A. The variation

$$n^{\text{name}^*} \text{ features of } i^{\text{inh}^*}. \tag{8.16}$$

declares that the set of names n is the defined and inherited features due to the inheritance clauses i. We define a function $\mathfrak{E}$ that extracts the names from a list of features.

$$\mathfrak{E}(\text{feat} \left[\ ?, n \ldots\ \right], l) = \{n\} \cup \mathfrak{E}(l) \tag{8.17}$$

$$\mathfrak{E}(\epsilon) = \emptyset \tag{8.18}$$

With this function, the rules for the judgment can be defined as:

$$\frac{\Gamma \vdash \text{cls} \left[\ ?, n, c\ \right] \qquad \Gamma \vdash b \text{ features of } c \triangledown \text{inh.}}{\Gamma \vdash \mathfrak{E}(c \triangledown \text{feat}) \cup b \text{ features of } n^{\text{name}}.}$$

$$\frac{\Gamma \vdash a \text{ features of } p.}{\Gamma \vdash \mathfrak{A}^c(a) \text{ features of inh} \left[\ \text{type} \left[\ p \ldots\ \right], c\ \right], l.}$$

The function $\mathfrak{A}$ in the judgment above *adapts* a syntax tree according to inheritance sub-clauses. With a list of names, it only applies renaming. The definition of this function is given in section 8.5.1.

$$\frac{}{\Gamma \vdash \emptyset \text{ features of } \epsilon.}$$

8.3.5 Properties of features

We need to distinguish attributes and once features from other features, as they have different semantics. This is stated by the following three judgments:

$$\{A\}\,.f \text{ is attribute.} \tag{8.19}$$

$$\{A\}\,.f \text{ is once.} \tag{8.20}$$

$$\{A\}\,.f \text{ is other.} \tag{8.21}$$

This can be achieved by looking at the clauses of the feature:

$$\frac{\Gamma \vdash \text{feat} \left[\ ?, f, b\ \right] \text{ is feature } A \text{ of } f. \qquad b\triangledown\text{do,once,deferred} = \epsilon}{\Gamma \vdash \{A\}\,.f \text{ is attribute.}}$$

$$\frac{\Gamma \vdash \text{feat} \left[\ ?, f, b\ \right] \text{ is feature } A \text{ of } f. \qquad b\triangledown\text{do,deferred} \neq \epsilon}{\Gamma \vdash \{A\}\,.f \text{ is other.}}$$

$$\frac{\Gamma \vdash \text{feat} \left[\ ?, f, b\ \right] \text{ is feature } A \text{ of } f. \qquad b\triangledown\text{once} \neq \epsilon}{\Gamma \vdash \{A\}\,.f \text{ is once.}}$$

8.4 Conformance

We describe rules to establish conformance. We only need a very simple form of conformance: whether a class inherits from another class or not. We write

$$c_1 \lessdot c_2 \tag{8.22}$$

if class c_1 is conforming to class c_2, that is, c_1 equals c_2 or is a descendant. Non-conformance is similar:

$$c_1 \not\lessdot c_2 \tag{8.23}$$

These judgments are supported by the following rules.

Reflexivity

$$\frac{}{\Gamma \vdash n \lessdot n}$$

Transitivity

$$\frac{\Gamma \vdash n_1 \lessdot n_2 \qquad \Gamma \vdash n_2 \lessdot n_3}{\Gamma \vdash n_1 \lessdot n_3}$$

Direct descendants

Conformance can be shown by splitting the inheritance structure established by the `inherit` clauses according to the transitivity rule above and the following rule showing conformance for direct descendants.

$$\frac{\Gamma \vdash \text{cls} \left[\ ?, n_1, c\ \right] \text{ exists.} \qquad \text{inh[typ}[n_2 \ldots]\ldots] \in c}{\Gamma \vdash n_1 \lessdot n_2}$$

Non-conformance

Showing non-conformance is traversing the graph upwards, which is required since it is neither transitive nor reflexive. To achieve this, we allow the n_1 in the $n_1 \not\leq n_2$ judgment to be a list of inheritance clauses. This variation of the judgment is the consequence of the following rules:

$$\frac{\Gamma \vdash l \not\leq n \qquad \Gamma \vdash p \not\leq n}{\Gamma \vdash \text{inh} \left[\text{ type} \left[\, p \ldots \, \right] \ldots \right], l \not\leq n}$$

$$\frac{}{\Gamma \vdash \epsilon \not\leq n}$$

Non-conformance can thus be shown according to the following rule:

$$\frac{\Gamma \vdash \text{cls} \left[\, ?, n_1, c \, \right] \text{ exists.} \qquad \Gamma \vdash c \triangledown \text{inh} \not\leq n_2}{\Gamma \vdash n_1 \not\leq n_2}$$

8.5 Resolving Features

Resolving features in Eiffel is complex due to multiple inheritance. It can be split into two steps. First, the actual name of the feature in the dynamic type of the object is determined. Second, the corresponding implementation is looked up, together with the necessary adaptation for names and types. Fortunately, we are only concerned with resolving unqualified calls, that is, calls without an explicit target, so we can concentrate on these. For the first part, we need to prove that the particular feature name is along the selected path of inheritance. As part of this, we also need to adapt the feature name, so we start by defining adaptation of names.

8.5.1 Adaptation

It is not sufficient to just retrieve the feature node when resolving a feature. We need to adapt it to the context; an important part of this is renaming. Also, if a feature is undefined or redefined, its body is thrown away. Adaptation of types is handled separately. Note that this function is not sufficiently defined for adapting features for execution: we limited it to adapt only the parts we are interested in.

$$\mathbb{B} = \{\text{name}, \text{str}, \text{int}, \text{real}\} \tag{8.24}$$

$$\begin{aligned}\mathfrak{A}^c(\alpha[a], r) &= b, \mathfrak{A}^c(r) \\ &\iff \text{rename}\left[\ \alpha[a], b\ \right] \in c \wedge \alpha \in \mathbb{B}\end{aligned} \tag{8.25}$$

$$\begin{aligned}\mathfrak{A}^c(\alpha[a], r) &= \alpha[a], \mathfrak{A}^c(r) \\ &\iff \text{rename}\left[\ \alpha[a], ?\ \right] \notin c \wedge \alpha \in \mathbb{B}\end{aligned} \tag{8.26}$$

$$\mathfrak{A}^c(\alpha\left[\ a\ \right], r) = \alpha\left[\ \mathfrak{A}^c(a)\ \right], \mathfrak{A}^c(r) \iff \alpha \notin \mathbb{B} \cup \{\text{feat}\} \tag{8.27}$$

$$\begin{aligned}\mathfrak{A}^c(\text{feat}\left[\ v, f, b\ \right], r) &= \text{feat}\left[\ \mathfrak{A}^c(v, f, b)\ \right], \mathfrak{A}^c(r) \\ &\iff \text{undefine}\left[\ \mathfrak{A}^c(f)\ \right] \notin c \wedge \text{redefine}\left[\ \mathfrak{A}^c(f)\ \right] \notin c\end{aligned} \tag{8.28}$$

$$\begin{aligned}\mathfrak{A}^c(\text{feat}\left[\ v, f, b\ \right], r) &= \text{feat}\left[\ \mathfrak{A}^c(v, f, b \triangledown \text{var}, \text{req}, \text{ens})\ \right], \mathfrak{A}^c(r) \\ &\iff \text{undefine}\left[\ \mathfrak{A}^c(f)\ \right] \in c \vee \text{redefine}\left[\ \mathfrak{A}^c(f)\ \right] \in c\end{aligned} \tag{8.29}$$

$$\mathfrak{A}^c(\epsilon) = \epsilon \tag{8.30}$$

$$\overline{\mathfrak{A}^c}(g) = f \iff \mathfrak{A}^c(f) = g \tag{8.31}$$

The core of adaptation is the adapt function $\mathfrak{A}^c$. This function adapts its argument according to the clauses in c, which are the rename, export, redefine and undefine clauses. The select clause has no impact on adapting, it is solely used for resolving the correct descendant. The function is recursive and applies the rename clauses to all names as shown in eqs. (8.25) and (8.26). These two equations and eq. (8.27) use α as a placeholder for the node name to avoid creating a version of the equation for every affected AST node type. Note that eq. (8.25) only applies to name nodes, as it is not possible to put other basic nodes in a rename node. The nodes not affected by the replacement are the basic nodes $\mathbb{B}$ and the feat node, which is handled by eqs. (8.28) and (8.29). These remove the local, do, once and attribute clauses if the feature is undefined or redefined. Adaptation does not include the export status.

We also define $\overline{\mathfrak{A}^c}$ as the inverse function of $\mathfrak{A}^c$. This is only used to adapt names.

8.5.2 Resolving Features

Resolving features is giving an answer to the question: "If `f` is called with a target that is of static type **A** and of dynamic type **B**, which feature in **B** is executed?"

The answer to this question can be found in the program text, but is difficult to obtain due to renaming and selection. The right answer can only be proven true by showing that it is a descendant feature in **B** of the feature `f` and it is along the `selected` path.

Selection. Selection solves the problem of choosing the right version of a feature if the context is ambiguous. A class in Eiffel can inherit from multiple classes. Every feature of a class has a unique name that can be changed in descendant classes with renaming. Suppose that class **A** defines feature `do_a` and feature `do_t`. The latter calls the former. The classes **B** and **C** both inherit from **A** and rename and redefine feature `do_a` to `do_b` and `do_c` respectively. Class **C** inherits from both **A** and **B**. If feature `do_t` is with an object of type **D** as a target, which feature is called? The original feature in **A**, the one redefined in **B** or the one in **C**? To solve this problem, a programmer has to select one of either `do_b` or `do_c` in the **inherit** clause, which is then called in the case above.

As stated above, the programmer needs to select a feature only if there is an ambiguity. It is therefore possible that the inheritance path of the correct feature does not contain a **select** clause, or that another path does also contain one. This makes it difficult to prove directly that the given answer is correct since it involves proving that all other candidates are not selected.

However, we can avoid using universal quantification by asserting that at every point in the hierarchy, the chosen feature was the one selected or the last one remaining. For this, we defined several functions:

$$\mathfrak{M}^c(a, r) = a, \mathfrak{M}^c(r) \quad \iff \text{select} \left[a \right] \notin c \tag{8.32}$$
$$\mathfrak{M}^c(a, r) = \underline{a}, \mathfrak{M}^c(r) \quad \iff \text{select} \left[a \right] \in c \tag{8.33}$$
$$\mathfrak{M}^c(\epsilon) = \epsilon \tag{8.34}$$
$$\mathfrak{S}(a, r) = \mathfrak{S}(r) \quad \iff r \neq \epsilon \tag{8.35}$$
$$\mathfrak{S}(\underline{a}, r) = a, \epsilon \tag{8.36}$$
$$\mathfrak{S}(a, \epsilon) = a, \epsilon \tag{8.37}$$
$$\mathfrak{S}(\epsilon) = \epsilon \tag{8.38}$$
$$\tag{8.39}$$

$\mathfrak{M}^c$ marks the selected features in a list of names by underlining it.
$\mathfrak{S}$ returns the (first) marked feature or the last feature in the list if none are marked[4].

Type conversion

Due to inheritance, it is possible that a generic parameter of a class can be constrained further due to the context. For example, suppose a program contains the two classes in listing 8.1.

```
class A [T]

feature
    attr: T

end

class B [T <- IMMUTABLE]

inherit
    A[T]
    IMMUTABLE

end
```

Listing 8.1: Adapting generics in immutable types

It is obvious that class **A** cannot be immutable: The generic parameter **T** is unconstrained and may or may not be immutable, which violates the immutability rule for

[4]In a valid Eiffel programs it is only possible to mark no features if this list contains just one kind of name. This property could be used to check whether the programmer has to select a feature.

attributes. However, class **B** is valid, since it constrains the generic parameter **T** to classes that inherit from **IMMUTABLE**. An analysis that checks the immutability of `attr` in context of class **B** therefore needs to take into account the constraint: it needs to convert the type in **A** using information from **B**.

The function $\mathfrak{C}$ is essential in doing so:

$$\mathfrak{C}_c^{p^{\text{param}}}(t) \tag{8.40}$$

This function augments the type t in context of type c where p is the list of parameters of the class that c points to. The generic variables in t are replaced by the constraints inferred from context and class parameters, with unconstrained parameters defaulting to **ANY**.

To come back to the example above, the term $\mathfrak{C}^{\text{param}\left[\ \mathbf{T}\ \right]}_{\text{type}\left[\ \mathbf{A}, \mathbf{IMMUTABLE}\ \right]}(\text{type}\left[\ \mathbf{T}\ \right])$ would yield type $\left[\ \mathbf{IMMUTABLE}\ \right]$ as a result.

$\mathfrak{C}$ is defined as follows:

$$\mathfrak{C}_c^p(\text{type}\left[\ n^{\text{name}}, p\ \right], l) = \text{type}\left[\ n, \mathfrak{C}_c^p(p)\ \right], \mathfrak{C}_c^p(l) \iff \text{param}\left[\ n\ldots\ \right] \notin p \tag{8.41}$$

$$\mathfrak{C}_c^p(\text{type}\left[\ n^{\text{name}}, \epsilon\ \right], l) = \mathfrak{L}_c^p(n), \mathfrak{C}_c^p(l) \iff \text{param}\left[\ n\ldots\ \right] \in p \tag{8.42}$$

$$\mathfrak{C}_c^p(\epsilon) = \epsilon \tag{8.43}$$

$$\mathfrak{L}^{\text{param}\left[\ n\ldots\ \right], l}_{\text{type}\left[\ ?, t\ldots\ \right]}(n) = t \tag{8.44}$$

$$\mathfrak{L}^{\text{param}\left[\ n\ldots\ \right], l}_{\text{type}\left[\ c, t, r\ \right]}(m) = \mathfrak{L}^{l}_{\text{type}\left[\ c, r\ \right]}(m) \iff n \neq m \tag{8.45}$$

The function $\mathfrak{C}$ complements a list of types given the context type and class and uses $\mathfrak{L}_c^p(n)$ to look up the corresponding field in the context type c based on the position of the name n in the list of generic parameters p.

Judgments

Finally, we can put everything together. We say that a certain call resolves to the following feature name by writing

$$g \text{ is } \{A\}.f \text{ in } B. \tag{8.46}$$

This judgment states that a call to f of class A on an object of type B is directed to the feature in B with the name g. A variant of the judgment operates on **inherit** clauses as B instead of a type, and a list of feature names instead of f.

Rules

$$\frac{}{\Gamma \vdash f \text{ is } \{A\}.f \text{ in type}\left[\ A\ldots\ \right].}$$

For the recursive step, the answer is the selected feature in the list g, which can be obtained by deriving $\mathfrak{M}^c$ for all adapted **inherit** clauses of B.

$$\frac{\Gamma \vdash \text{cls}\left[\ ?, B, c\ \right] \text{ exists.} \qquad \Gamma \vdash g \text{ is } \{A\}.f \text{ in } c\triangledown\text{inh}.}{\Gamma \vdash \mathfrak{S}(g) \text{ is } \{A\}.f \text{ in type}\left[\ B\ldots\ \right].}$$

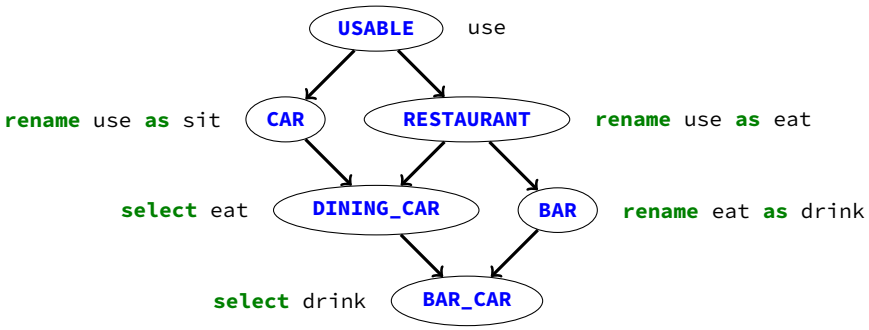


Figure 8.3: Resolving features example: classes

$$\frac{\Gamma \vdash r \text{ is } \{A\}.f \text{ in } i. \qquad \Gamma \vdash g \text{ is } \{A\}.f \text{ in } B.}{\Gamma \vdash \mathfrak{M}^c(\mathfrak{A}^c(g)), r \text{ is } \{A\}.f \text{ in inh } [\text{ type } [\ B\ldots\ ]\ , c\ ]\ , i.}$$

$$\frac{}{\Gamma \vdash \epsilon \text{ is } \{?\}.? \text{ in } \epsilon.}$$

8.5.3 Example

The graph in fig. 8.3 shows six classes forming two diamonds. The central feature of the example is `{USABLE}.use`, which is being renamed in **CAR**, **RESTAURANT** and **BAR**. The **CAR** represents a train car, and a **BAR** is a variant of a **RESTAURANT** which primarly serves drinks. There are two variations of cars in addition to the basic one: one for dining and one for drinking. This example is artificial, its purpose is solely to explain how selection works without introducing too many features and classes.

The proof trees in fig. 8.4 establish which effective feature is called if the feature `use` is invoked on a target of static type **USABLE** representing an object of any of the dynamic types in fig. 8.3. Note that we omitted the leafs proving existence, that is, judgments ending in *exists.*, to fit the proofs within one page.

$$\frac{\Gamma \vdash \epsilon \text{ is } \{\texttt{USABLE}\}.\texttt{use in } \epsilon. \qquad \Gamma \vdash \texttt{use is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{USABLE}\ ].}{\dfrac{\Gamma \vdash \texttt{sit is } \{\texttt{USABLE}\}.\texttt{use in inh } [\ \text{type } [\ \texttt{USABLE}\ ], \text{rename } [\ \texttt{use}, \texttt{sit}\ ]\ ].}{\Gamma \vdash \texttt{sit is } \{\texttt{USABLE}\}.\texttt{use in } \texttt{CAR}.}}$$

$$\frac{\Gamma \vdash \epsilon \text{ is } \{\texttt{USABLE}\}.\texttt{use in } \epsilon. \qquad \Gamma \vdash \texttt{use is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{USABLE}\ ].}{\dfrac{\Gamma \vdash \texttt{eat is } \{\texttt{USABLE}\}.\texttt{use in inh } [\ \text{type } [\ \texttt{USABLE}\ ], \text{rename } [\ \texttt{use}, \texttt{eat}\ ]\ ].}{\Gamma \vdash \texttt{eat is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{RESTAURANT}\ ].}}$$

$$\frac{\Gamma \vdash \epsilon \text{ is } \{\texttt{USABLE}\}.\texttt{use in } \epsilon. \qquad \Gamma \vdash \texttt{eat is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{RESTAURANT}\ ].}{\dfrac{\Gamma \vdash \texttt{drink is } \{\texttt{USABLE}\}.\texttt{use in inh } [\ \text{type } [\ \texttt{RESTAURANT}\ ], \text{rename } [\ \texttt{eat}, \texttt{drink}\ ]\ ].}{\Gamma \vdash \texttt{drink is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{BAR}\ ].}}$$

$$\frac{\dfrac{\Gamma \vdash \epsilon \text{ is } \{\texttt{USABLE}\}.\texttt{use in } \epsilon. \qquad \Gamma \vdash \texttt{eat is } \{\texttt{USABLE}\}.\texttt{use in } \texttt{RESTAURANT}.}{\Gamma \vdash \underline{\texttt{eat}} \text{ is } \{\texttt{USABLE}\}.\texttt{use in inh } [\ \text{type } [\ \texttt{RESTAURANT}\ ], \text{select } [\ \texttt{eat}\ ]\ ].} \qquad \Gamma \vdash \texttt{sit is } \{\texttt{USABLE}\}.\texttt{use in } \texttt{CAR}.}{\dfrac{\Gamma \vdash \texttt{sit}, \underline{\texttt{eat}} \text{ is } \{\texttt{USABLE}\}.\texttt{use in inh } [\ \text{type } [\ \texttt{CAR}\ ]\ ], \text{inh } [\ \text{type } [\ \texttt{RESTAURANT}\ ], \text{select } [\ \texttt{eat}\ ]\ ].}{\Gamma \vdash \texttt{eat is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{DINING\_CAR}\ ].}}$$

$$\frac{\dfrac{\Gamma \vdash \epsilon \text{ is } \{\texttt{USABLE}\}.\texttt{use in } \epsilon. \qquad \Gamma \vdash \texttt{drink is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{BAR}\ ].}{\Gamma \vdash \underline{\texttt{drink}} \text{ is } \{\texttt{USABLE}\}.\texttt{use in inh } [\ \text{type } [\ \texttt{BAR}\ ], \text{select } [\ \texttt{drink}\ ]\ ].} \qquad \Gamma \vdash \texttt{eat is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{DINING\_CAR}\ ].}{\dfrac{\Gamma \vdash \texttt{eat}, \underline{\texttt{drink}} \text{ is } \{\texttt{USABLE}\}.\texttt{use in inh } [\ \text{type } [\ \texttt{DINING\_CAR}\ ]\ ], \text{inh } [\ \text{type } [\ \texttt{BAR}\ ], \text{select } [\ \texttt{drink}\ ]\ ].}{\Gamma \vdash \texttt{drink is } \{\texttt{USABLE}\}.\texttt{use in type } [\ \texttt{BAR\_CAR}\ ].}}$$

Figure 8.4: Resolving features examples: proof trees

8.6 Immutability as a Property of Classes

There are three different levels at which it is possible to specify that objects are immutable:

1. At the object level, for example at creation time
2. At the type level, for example as a type modifier
3. At the class level, for example as a marker

If we want to reason statically about immutable objects, then the first option is not practical. Reasoning at the type level is possible and many related approaches, some of them discussed in section 8.12, go this route. However, as we will see soon, immutability puts a variety of requirements on the class structure, so specification at the class level is most convenient. It also avoids overloading the type system with even more modifiers. In fact, it only requires the introduction of a special **IMMUTABLE** class and no further annotation.

To obtain a fully formal inductive reasoning system for proving classes immutable, which can also be automated for use within a compiler, we need to specify what immutability means and what its consequences are. We start by stating the general property of immutable objects:

Property 1. *An object of an immutable class may not change its state, that is, the value of its attributes, after the creation routine is finished. We write*

$$C \textit{ immutable.} \tag{8.47}$$

to declare class C is immutable.

This property implies that it is permissible to change the state during creation, even if the creation routine calls other routines to modify the state.

Listing 8.2 shows a problem: feature `do_something` in class **UNSUSPECTING** requires an argument of type **IMMU_CLASS**, of which it rightfully assumes that it is immutable. However, the client may pass an instance of **MUT_CLASS** instead and then modify the state after the supplier routine finished. We therefore conclude:

Property 2. *Classes inheriting from immutable classes must be immutable.*

This property also implies how immutable classes can be marked:

Property 3. *Immutable classes inherit from **IMMUTABLE**.*

We now have the general conditions for declaring a class immutable, which we formalize as the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash f \text{ features of } C.\\ \Gamma \vdash \text{cls}\left[\ ?, C, p\ \right] \text{ exists.}\\ \Gamma \vdash f \text{ in } \quad \text{safe.}\\ \Gamma \vdash \text{all invariants of } \mathfrak{I}_C(p\triangledown\text{param}) \text{ in context } \mathfrak{I}_C(p\triangledown\text{param}) \text{ immutating.}\end{array}}{\Gamma \vdash C \text{ immutable.}}$$

We use the term immutating for instructions and features that are not mutating immutable state. The rule specifies that all the features of an (existing) immutable class need to be safe, which means that they adhere to the immutability constraints. The class needs to inherit from **IMMUTABLE**, and all its invariants need to be immutating, which we will explain later.

```
class IMMU_CLASS
create
    make
feature {NONE}
    make (a: ANY)
        do
            item := a
        end
feature
    item: ANY
end

class MUT_CLASS
inherit
    IMMU_CLASS
feature
    set_item (a: ANY)
        do
            item := a
        end
end

class UNSUSPECTING
feature
    do_something (immu_object: IMMU_CLASS)
        deferred
        end
end
```

Listing 8.2: Mutable class inheriting from immutable class

8.6.1 Mutable Classes

A system may contain immutable and mutable classes. Mutable classes have no restrictions, except for the fact that

Property 4. *Mutable classes may not inherit from* ***IMMUTABLE****. We write*

$$A \text{ mutable.} \tag{8.48}$$

to express that the class A is mutable

Establishing mutability is done by this simple rule:

$$\frac{\Gamma \vdash A \not\leq \texttt{IMMUTABLE}}{\Gamma \vdash A \text{ mutable.}}$$

8.7 Mutating Features

8.7.1 Safe Features

The next step is to define when a feature adheres to the immutability rules.

Property 5. *All features of an immutable class need to preserve the immutability of the object. We call such features safe and we write*

$$f \textit{ in } A \textit{ safe.} \tag{8.49}$$

if the feature name or list of feature names f are safe in class A

We can show safety individually, so we introduce the following rules

$$\frac{\Gamma \vdash f \text{ in } a \text{ safe.} \qquad \Gamma \vdash l \text{ in } a \text{ safe.}}{\Gamma \vdash f, l \text{ in } a \text{ safe.}}$$

$$\frac{}{\Gamma \vdash \epsilon \text{ in } ? \text{ safe.}}$$

8.7.2 Private Features

A feature that cannot be called after the construction can never modify the state. The constructor, or any features it uses, if exported to onl **NONE** are permissible. While it is important that these features are not called, directly or indirectly, from features accessible by clients other than **Current**, they are permissible as safe. We call these features private.

Property 6. *A feature that can only be called from the same object is private. To state that a feature with name f in class A is private we write:*

$$f \textit{ in } A \textit{ private.} \tag{8.50}$$

A variation of the judgment uses instead of a class A a list of feature and inheritance clauses, in which case the informal meaning is that the symbol f is private in all the classes denoted in the inheritance and all the features. This enables us to show privateness inductively through the following rules:

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, A, c\ \right] \text{ exists.} \\ \Gamma \vdash f \text{ in } c\triangledown\text{inh private.} \\ \Gamma \vdash \text{feat} \left[\ \text{visib} \left[\ \textbf{NONE}\ \right] \ldots\ \right] \in c\end{array}}{\Gamma \vdash f^{\text{name}} \text{ in } A^{\text{name}} \text{ private.}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, A, c\ \right] \text{ exists.} \\ \Gamma \vdash f \text{ in } c\triangledown\text{inh private.} \\ f \notin \mathfrak{E}(c\triangledown\text{feat})\end{array}}{\Gamma \vdash f^{\text{name}} \text{ in } A^{\text{name}} \text{ private.}}$$

$$\frac{\Gamma \vdash f \text{ in } l \text{ private.} \qquad \Gamma \vdash \overline{\mathfrak{A}^c}(f) \text{ in } P \text{ private.}}{\Gamma \vdash f^{\text{name}} \text{ in inh} \left[\ \text{type} \left[\ P \ldots\ \right], c\ \right], l \text{ private.}}$$

$$\frac{}{\Gamma \vdash ? \text{ in } \epsilon \text{ private.}}$$

We show that a feature is private by induction. At each step we show that the feature denoted by the given name is either not defined or it is defined with a visibility of **NONE** in a class. We then inductively show the same for all the ancestor classes, for which we reversely adapt the feature name according to the inheritance clause. Due to polymorphism, it does not suffice to show that the feature is not exported in the immutable class itself.

A safe feature can be private and immutating: this is necessary if the feature is indirectly accessible from a client other than **Current**. In our reasoning system, we say that if a feature is private, this implies that it is safe.

$$\frac{\Gamma \vdash f^{\text{name}} \text{ in } A^{\text{name}} \text{ private.}}{\Gamma \vdash f^{\text{name}} \text{ in } A^{\text{name}} \text{ safe.}}$$

8.8 Immutating Features

Property 7. *Features are safe if they are not mutating the state of the current object, that is, if they are* immutating. *We write*

$$\frac{\theta \in \{\textit{regular, deferred}\}}{f \textit{ in } a^{\textit{type}} \textit{ context } c^{\textit{type}} \ \theta \textit{ immutable.}} \tag{8.51}$$

to state that a feature f of tpye a in context of type c is immutating.

Note that this does not mean that it is free of side effects: it may, for example, modify arguments. The variable θ with either *regular* or *deferred* as its value indicates whether the complete feature is checked (*regular*) or just the interface (*deferred*). The latter is weaker, as it specifies that the feature is immutable without consideration of its body, that is, as if it was deferred. Before we can derive a rule from this property, we need a function that creates the initial context. $\mathfrak{I}$ takes the parameters of a class including the constraints and generates a type that consists of the constraints, or **ANY** if unconstrained.

$$\mathfrak{I}_c(p) = \text{type} \left[\ c, \mathfrak{I}(p)\ \right] \tag{8.52}$$

$$\mathfrak{I}(\text{param} \left[\ n\ \right], l) = \textbf{ANY}, \mathfrak{I}(l) \tag{8.53}$$

$$\mathfrak{I}(\text{param} \left[\ n, c\ \right], l) = c, \mathfrak{I}(l) \tag{8.54}$$

$$\mathfrak{I}(\epsilon) = \epsilon \tag{8.55}$$

From the property and the function we conclude the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash f \text{ in } \mathfrak{I}_c(p \triangledown \text{param}) \text{ context } \mathfrak{I}_c(p \triangledown \text{param}) \text{ regular immutable.} \\ \Gamma \vdash \text{cls} \left[\ ?, A, c\ \right] \text{ exists.}\end{array}}{\Gamma \vdash f^{\text{name}} \text{ in } A^{\text{name}} \text{ safe.}}$$

8.8.1 Ancestors

However, establishing that a feature is immutating also involves the ancestors of the feature, similarly to privateness. We use the judgment

$$\frac{\theta \in \{\text{regular, deferred}\}}{f \text{ in } i^{\text{inh}^*} \text{ of } a^{\text{type}} \text{ context } c^{\text{type}} \ \theta \text{ immutable.}} \tag{8.56}$$

to assert that the ancestors of feature f due to inheritance clauses i of a in context of c are immutating. This judgment is derived using these two rules:

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, n, c\ \right] \text{ exists.} \\ \Gamma \vdash \overline{\mathfrak{A}^i}(f) \text{ in } \mathfrak{C}^{c\triangledown\text{param}}_{\text{type}\left[\ n,p\ \right]}(\text{type} \left[\ a, g\ \right]) \text{ context } t\ \theta' \text{ immutable.} \\ \Gamma \vdash f \text{ in } l \text{ of type } \left[\ n,p\ \right] \text{ context } t\ \theta \text{ immutable.} \\ \theta' = \begin{cases} \theta \iff \text{redefine} \left[\ f\ \right] \notin i \wedge \text{undefine} \left[\ f\ \right] \notin i \\ \text{deferred} \end{cases}\end{array}}{\Gamma \vdash f \text{ in inh} \left[\ \text{type} \left[\ a, g\ \right], i\ \right], l \text{ of type } \left[\ n,p\ \right] \text{ context } t\ \theta \text{ immutable.}}$$

$$\frac{}{\Gamma \vdash ? \text{ in } \epsilon \text{ of } ? \text{ context } ?\ ? \text{ immutable.}}$$

8.8.2 Undefined Features

A feature that is not defined in a class is immutating if its ancestors are. We express this with the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, n, c\ \right] \text{ exists.} \\ f \notin \mathfrak{E}(c\triangledown\text{feat}) \\ \Gamma \vdash f \text{ in } c\triangledown\text{inh of type } \left[\ n,p\ \right] \text{ context } t\ \theta \text{ immutable.}\end{array}}{\Gamma \vdash f \text{ in type } \left[\ n,p\ \right] \text{ context } t\ \theta \text{ immutable.}}$$

8.8.3 Attributes

What makes a feature immutating depends on the type of the feature. We start by looking at attributes as the state of a class. Since attributes cannot change by themselves, a first intuitive approach is to ignore them.

However, listing 8.3 reveals a problem with this approach. Obviously the class in this listing is immutable according to the specification given so far: its attribute is set during creation and not modified afterwards. However, it is not what a client expects from an immutable object, since it does seem to have a mutable state, even if it is encapsulated. To satisfy this expectation we further specify:

Property 8. *Attributes of an object of an immutable class may only refer to or contain objects of an immutable class, or must be declared separate.*

This ensures that everything that can be reached from an immutable class is immutable itself or separate. This is important since we want to be able to access immutable objects non-separate in all processors. If we would allow (non-separate) references to mutable classes, we would introduce traitors. Whether the object referenced by a separate attribute is separate depends on the processor that is executing the code. Attributes of generic type need also be either of immutable or separate type.

This property is formally expressed with the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, n, c\ \right] \text{ exists.} \\ \Gamma \vdash \text{feat} \left[\ ?, f, b\ \right] \text{ is feature } f \text{ of } n. \\ \Gamma \vdash f \text{ in } c\triangledown\text{inh of type } \left[\ n,p\ \right] \text{ context } t\ \theta \text{ immutable.} \\ \Gamma \vdash \{n\}.f \text{ is attribute.} \\ \Gamma \vdash \texttt{separate} \in b \vee \mathfrak{C}^{c\triangledown\text{param}}_{\text{type}\left[\ n,p\ \right]}(\mathfrak{H}(b\triangledown\text{type})) \lessdot \texttt{IMMUTABLE}\end{array}}{\Gamma \vdash f \text{ in type } \left[\ n,p\ \right] \text{ context } t\ \theta \text{ immutable.}}$$

```
class FAKE_IMMU_CLASS

create make

feature {NONE} make
    do
        create t
    end

feature {NONE} t: CELL[ANY]

feature item: ANY
    do
        Result := t.item
    end

feature set_item (a_item: ANY)
    do
        t.set_item (a_item)
    end
end
```

Listing 8.3: Immutable class wrapping mutable object

An attribute is immutable if its ancestors are immutable and its type is immutable.

8.8.4 Routines

For routines, that is procedures and functions, the result type does not matter. However, to be considered immutating, routines have effects that need to be restricted:

Property 9. *An immutating feature does not assign to attributes.*

This property is not enough, since features can call other features. We therefore conclude that

Property 10. *An immutating feature does not make unqualified calls to mutating features.*

From property 5 we can gather that all qualified calls to an immutable object are immutating; this includes calls with `Current` as a target according to the rules of Eiffel. Note that calls are also part of the pre- and postcondition, not just the body of a routine.

For normal routines, the rule is therefore:

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, n, c\ \right] \text{ exists.} \\ \Gamma \vdash \text{feat} \left[\ ?, f, b\ \right] \text{ is feature } f \text{ of } n. \\ \Gamma \vdash b \text{ local } \epsilon \text{ in type } \left[\ n, p\ \right] \text{ context } t\ \theta \text{ immutable.} \\ \Gamma \vdash f \text{ in } c \triangledown \text{inh of type } \left[\ n, p\ \right] \text{ context } t\ \theta \text{ immutable.} \\ \Gamma \vdash \{n\}\,.f \text{ is other.}\end{array}}{\Gamma \vdash f^{\text{name}} \text{ in type } \left[\ n, p\ \right] \text{ context } t\ \theta \text{ immutable.}}$$

The rule requires that all the instructions and clauses of the feature are immutating, which we cover in section 8.9.

8.8.5 Once Features

While the above properties cover most features, they are not sufficient for the **once** routines. These routines are special because their intrinsic value is, with exception of the object once routine, not bound to the object but the class.

In general, these routines can be deemed mutating since they change state. In the case of procedures, the only state is an indication of whether they have been run before or not, while in the case of functions it also encompasses their result. For these reasons, it seems that once routines should simply be banned from appearing in immutable classes. However, this would be too restrictive and may cause problems because the root class **ANY** already contains once routines. We therefore choose to allow once routines, but restrict them in the same way as regular routines, with the following justification:

The scope of process once routines is the whole process, which, in D-Scoop, is the current node: there are no system wide once routines in D-Scoop. As such, they are not in the scope of the immutability property of the object. A similar reasoning can be applied to thread-once routines: they are thread-local routines and variables and not specific to an object. The thread scope is processor local, so such a once routine would keep the state per processor. For object-once routines, the reasoning is different: an object-once routine is effectively just a lazily initialized attribute, so we consider it to be a deferred part of the creation routine. However, if the immutable object is copied by the D-Scoop runtime before this attribute has been initialized, it may be initialized to different values on different nodes. It is a matter of debate whether the runtime should prevent that in some way.

In all of these cases, a once routine may not change the state of the object through assignments or calls. Since the value of once functions is determined by the special, locally bound **Result** symbol, there does not need to be any exception for once functions in the rules for features. However, object-once functions are treated like attributes:

Property 11. *The type of object-once functions needs to be immutable.*

Note that for the usage in Scoop, once-features of the same object may not give the same result when called from different processors. In D-Scoop with transparent copying, this is also the case for object-once functions if they have not been called before the object was copied.

The rules for once features are therefore:

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, n, c\ \right] \text{ exists.} \\ \Gamma \vdash \text{feat} \left[\ ?, f, r^{\text{type}}, b\ \right] \text{ is feature } f \text{ of } n. \\ \Gamma \vdash b \text{ local } \epsilon \text{ in type} \left[\ n, p\ \right] \text{ context } t\ \theta \text{ immutable.} \\ \Gamma \vdash f \text{ in } c\triangledown\text{inh of type} \left[\ n, p\ \right] \text{ context } t\ \theta \text{ immutable.} \\ \Gamma \vdash \{n\}\ .f \text{ is once.} \\ r = \mathfrak{H}(b\triangledown\text{type}) \\ \text{once} \left[\ \text{oscope} \left[\ \texttt{"OBJECT"}\ \right] \ldots\ \right] \notin b \vee r = \epsilon \vee \mathfrak{C}^{c\triangledown\text{param}}_{\text{type}\left[\ n, p\ \right]}(r) \lessdot \textbf{IMMUTABLE}\end{array}}{\Gamma \vdash f \text{ in type} \left[\ n, p\ \right] \text{ context } t\ \theta \text{ immutable.}}$$

8.8.6 Invariants

Although invariants are not features, they can be seen as part of all the features since they are checked before and after every qualified call. This leads us to the next property:

Property 12. *Invariants of an immutable class may only call immutable features. We state that the invariants of a class a in context of c are immutating by writing:*

$$\textit{all invariants of } a^{\textit{type}} \textit{ in context } c^{\textit{type}} \textit{ immutating.} \tag{8.57}$$

Although an invariant, or any other type of assertion, that mutates the state of a class is certainly bad practice[5], it is technically allowed by Eiffel. The judgment on the immutating aspect of invariants is a simplified form of the judgments for features. It simply states that the invariants of a class in a specific context are immutating. There is no need to identify each invariant and there is also no local context. To derive a rule, we define an an additional judgment for stating that the invariants of all ancestors are immutating:

$$\text{invariants } i^{\text{inh}^*} \text{ of } a^{\text{type}} \text{ in context } c^{\text{type}} \text{ immutating.} \tag{8.58}$$

The rules are similar to the ones for features:

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, n, c\ \right] \text{ exists.} \\ \Gamma \vdash c \triangledown \text{inv local } \epsilon \text{ in type} \left[\ n, p\ \right] \text{ context } c\ \theta \text{ immutable.} \\ \Gamma \vdash \text{all invariants of } c \triangledown \text{inh in context } c \text{ immutating.}\end{array}}{\Gamma \vdash \text{all invariants of type} \left[\ n, p\ \right] \text{ in context } c \text{ immutating.}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \text{cls} \left[\ ?, n, b\ \right] \text{ exists.} \\ \Gamma \vdash \text{all invariants of } \mathfrak{C}^{b \triangledown \text{param}}_{\text{type}\left[\ n, p\ \right]}(t) \text{ in context } c \text{ immutating.} \\ \Gamma \vdash \text{invariants } i \text{ of type} \left[\ n, p\ \right] \text{ in context } c \text{ immutating.}\end{array}}{\Gamma \vdash \text{invariants inh} \left[\ t \ldots\ \right], i \text{ of type} \left[\ n, p\ \right] \text{ in context } c \text{ immutating.}}$$

$$\frac{}{\Gamma \vdash \text{invariants } \epsilon \text{ of } ? \text{ in context } ? \text{ immutating.}}$$

8.9 Instructions and Expressions

Checking for immutability of the contents of features needs to include an additional context: local variables. The following judgment is used to make a statement about instructions, feature clauses and expressions. It incorporates a list of names that are local and can be safely queried and assigned to. Note that variable a can assume many different types of nodes and node lists.

$$\frac{\theta \in \{\text{regular, deferred}\}}{a \text{ local } l^{\text{name}^*} \text{ in } a^{\text{type}} \text{ context } c^{\text{type}}\ \theta \text{ immutable.}} \tag{8.59}$$

The rules to ensure immutability of feature clauses, instructions and expressions, are numerous but simple, so give them here without detailed comments.

[5] Assertions may be removed from optimized binaries, and mutating state in a query is a violation of the command/query separation principle.

8.9.1 Feature clauses

$$\frac{\Gamma \vdash r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.} \qquad i \wr \text{ args, type, attribute, deferred}}{\Gamma \vdash i, r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

$$\frac{\Gamma \vdash r \text{ local } v + l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}{\Gamma \vdash \text{local } [\ v\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

$$\frac{\Gamma \vdash r \text{ local } n, l \text{ in } c \text{ context } \theta \quad \text{immutable.}}{\Gamma \vdash \text{var } [\ n, t\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

$$\frac{\Gamma \vdash e, r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.} \qquad \alpha \in \{ \text{ req, ens } \}}{\Gamma \vdash \alpha\ [\ e\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

$$\frac{\Gamma \vdash b, r \text{ local } l \text{ in } q \text{ context } c \text{ regular immutable.}}{\Gamma \vdash \text{do } [\ b\ ], r \text{ local } l \text{ in } q \text{ context } c \text{ regular immutable.}}$$

$$\frac{\Gamma \vdash r \text{ local } l \text{ in } q \text{ context } c \text{ deferred immutable.}}{\Gamma \vdash \text{do } [\ \ldots\ ], r \text{ local } l \text{ in } q \text{ context } c \text{ deferred immutable.}}$$

$$\frac{\Gamma \vdash b, r \text{ local } l \text{ in } q \text{ context } c \text{ regular immutable.}}{\text{once } [\ ?, b\ ], r \text{ local } l \text{ in type } [\ n, p\ ] \text{ context } c \text{ regular immutable.}}$$

$$\frac{\Gamma \vdash r \text{ local } l \text{ in } q \text{ context } c \text{ deferred immutable.}}{\Gamma \vdash \text{once } [\ \ldots\ ], r \text{ local } l \text{ in } q \text{ context } c \text{ deferred immutable.}}$$

8.9.2 Conditionals and Loops

$$\frac{\begin{array}{c}\Gamma \vdash b, r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.} \\ \alpha \in \{ \text{ cond, then, elseif, else, from, until, vari, inv, loopb } \}\end{array}}{\Gamma \vdash \alpha\ [\ b\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

Note that the rule above is also used for class invariants, in which case list l is always empty.

8.9.3 Feature calls

A qualified feature call is always safe: even if the target is **`Current`**, the call is only valid if the feature is exported. However, the arguments need to be immutable.

$$\frac{\Gamma \vdash t, a + r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}{\Gamma \vdash \text{qc } [\ \text{target } [\ t\ ] \text{ name } [\ f\ ]\ a\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

An unqualified feature call is only immutable if the target feature is immutable.

$$\frac{\begin{array}{c}\Gamma \vdash a + r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.} \\ \Gamma \vdash f \text{ in } q \text{ context } c \text{ regular immutable.}\end{array}}{\Gamma \vdash \text{uc } [\ \text{name } [\ f\ ]\ a\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

8.9.4 Assignment

Assignment is only immutable if it is to a local variable or to the special **`Result`** variable. Note that assignment to arguments and to scoped variables, such as the the ones introduces with object tests, are invalid in Eiffel.

$$\frac{\Gamma \vdash e, r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.} \qquad n \in l \cup \{\texttt{Result}\}}{\Gamma \vdash \text{assign } [\ n, e\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

8.9.5 Creation instruction

Creation is immutable if the target is a local variable.

$$\frac{\Gamma \vdash a + r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.} \qquad n \in l \cup \{\texttt{Result}\}}{\Gamma \vdash \text{createi } [\ n, f, a\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

8.9.6 Expressions

The **`create`** expression does not assign to a variable, it is therefore immutable if the arguments are immutable.

$$\frac{\Gamma \vdash a + r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}{\Gamma \vdash \text{createe } [\ n, f, a\ ], r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

Constants are immutable.

$$\frac{\Gamma \vdash r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.} \qquad k \wr \text{ char, str, int, real, bool}}{\Gamma \vdash k^{\text{char}}, r \text{ local } l \text{ in } q \text{ context } c\ \theta \text{ immutable.}}$$

8.10 Adherence of Programs to Immutability

What remains to be addressed is how we prove that a program adheres to immutability rules. This is expressed by the property that

Property 13. *A valid program consists of mutable and immutable classes only. We state that a program p is valid by writing*

$$p \text{ ok.} \tag{8.60}$$

The inference rules to establish validity are straight forward:

$$\frac{p \text{ exists.} \vdash p \text{ ok.}}{p^{\text{prog}}}$$

$$\frac{\Gamma \vdash c \text{ ok.} \qquad \Gamma \vdash p \text{ ok.}}{\Gamma \vdash c^{\text{cls}} p^{\text{prog}} \text{ ok.}}$$

$$\frac{\Gamma \vdash c \text{ mutable.}}{\Gamma \vdash c^{\text{cls}} \text{ ok.}}$$

$$\frac{\Gamma \vdash c \text{ immutable.}}{\Gamma \vdash c^{\text{cls}} \text{ ok.}}$$

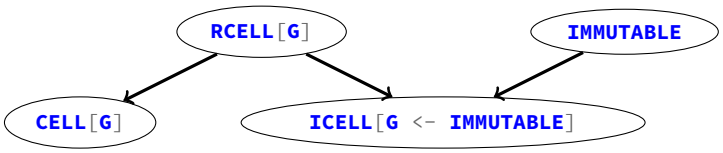


Figure 8.5: Cells example: inheritance

```
class RCELL[G]

create
    make

feature {NONE}
    make (a_item: G)
        do
            set_item (a_item)
        end

    set_item (a_item: G)
        do
            item := a_item
        end

feature
    item: G

end
```

```
class CELL[G]

create
    make

inherit
    RCELL[G]
        export {ANY}
            set_item
        end

end

class ICELL[G <- IMMUTABLE]

create
    make

inherit
    RCELL[G]
    IMMUTABLE

end
```

Listing 8.4: Cells example: classes

8.11 Example

As an example for using the proof system we present an immutable cell class. While the usefulness of an immutable cell is debatable, it explains two concepts. First, it shows how to conform to a generic class and preserve immutability and second, it shows how to provide both immutable and mutable versions of a class.

The conformance problem is solved by constraining the generic parameter to IMMUTABLE, thus making the attribute immutable which allows it to conform to the restrictions of immutability.

As for the second, this is something that is often encountered: A version of the class that only provides features for immutable access and a version that also allows mutation. A common scheme is to make the immutable version of the class the parent of the mutable one: since mutability adds functionality, this is reasonable. However, immutability as a property should be inheritable as established in section 8.6, so this is not an option. The solution is to use a common parent, the *readable* version, to both the mutable and the immutable version of the class, as shown in fig. 8.5. This allows

the usage of the common class whenever only readable access is needed, and of the two other other classes if their specific characteristic is needed.

8.12 Related Work

Immutability is an inherent property of functional languages due to their statelessness, and is replicated in many programming languages as an unchecked property of some classes, for example String in Java. The approach most similar to ours is [68], which, although is does not change the syntax of Java, does require a great amount of annotation, but also allows for immutable references. An approach using ownership types is [24]. While this gives some more flexibility, especially the fact that it can operate with classes that are not checked, it does come at the cost of a great amount of annotation. Another approach that relaxes the requirement for immutable classes to be completely initialized after the constructor has run is [36]. This is based on static verification techniques and requires annotation such as modifies clauses. Approaches that are specifically treating reference immutability, as in [64], which ensure that references cannot be used to change an object, are not sufficient with our intention of using it for SCOOP and D-SCOOP.

8.13 Future Work

The current immutability checker is limited to be used with regular SCOOP since the D-SCOOP prototype does not yet support export of objects. In future work, we plan to integrate the support for immutable classes into D-SCOOP. Furthermore, an immutable variant of the `SPECIAL` class that backs arrays in Eiffel is needed to get true immutable strings. Implementing this and other basic immutable classes are another topic to work on in the future.

Chapter 9

Conclusion

This dissertation discusses a system for distributed and parallel computing based on the Scoop concurrency model. We introduce the area and identify the need for a solution that can be used to manage concurrency both on the level of intra-node processors and of inter-node communication. Current solutions lack either the universal applicability or do not offer sufficient safety and reliability features. To achieve our goal, we choose the Scoop model for its advanced mechanisms to avoid high level data races.

As a justification for the need of finding a unifying model as well as the comparable efforts in the field, chapter 2 gives an overview over the most prominent models suitable to fill the gap. In addition, this chapter defines a terminology for describing object-oriented concurrent and distributed models and languages. Such a definition is needed because of the many terms used in this area of research, some of them, such as active object, for multiple different concepts.

The model of choice, Scoop, is presented in some more detail in a dedicated chapter, chapter 3. While this introduction is not supposed to be a complete tutorial, it offers readers a complete overview of the mechanisms of Scoop and defines the important terms *controlled* and *locked* as well as the approach Scoop takes on exclusive access.

In order to support the claim that a distributed application of the Scoop model can fill the need for many different types of distributed applications, the examples in chapter 4 cover the three widely used communication architectures: peer-to-peer, client-server and manager-worker. They further demonstrate one of the key innovations of D-Scoop: compensation. A less obvious, but no less important feature of D-Scoop is shown by omission: that D-Scoop does not require any additional annotation compared to regular Scoop. This means that the transition from a regular Scoop program to a D-Scoop program can be done without significant modifications.

The dissertation gives an informal overview of the D-Scoop system and framework in chapter 5. The system works on top of a simple message passing protocol, and its most important aspect of the protocol is shown in detail: the prelock phase that ensures a proper ordering of requests. The seamless integration of the prelock phase into the Scoop model based on Queue-of-Queues using proxy processors ensures that there is no performance regression for purely local operations as well as no preferential treatment of local over remote clients. Furthermore, the chapter compares the performance of D-Scoop with the widely used network object model Java RMI and finds reassuring results: as soon as there is some synchronization needed,

the D-Scoop system can outperform RMI. Even in cases that are light on synchronizations, the D-Scoop system offers a well comparable performance.

To close the gap between the protocol and the Scoop model, this work includes a formal specification of the protocol and the D-Scoop semantics in chapter 6. The author reduces the complexity of the semantics drastically by choosing an abstraction that narrows the focus on the important aspects: the handling of locks, exceptions and compensation, as well as the interactions between processors. From the semantics, an implementor can deduce which information has to be stored at which part of a processor, while not prescribing a model for heap or stack memory.

The presented material supports our first hypothesis, which states that the Scoop model can be used as a reliable base for distributed programming. The remaining two core chapters are dedicated to the integration of shared memory in a model that is based on segregation of regions. Since shared memory is available on most modern computer architectures, not being able to take advantage of it is a major detractor to the usage of safe concurrency models such as Scoop. These chapters support the second hypothesis of the thesis.

The first approach, which is presented in chapter 7, is by providing an API that is designed to maintain all guarantees established by the Scoop model while at the same time internally circumvents the rules imposed by the model so that it can use shared memory whenever possible. The API focuses on array structures, but is generalizable to other divisible data structures such as hash tables or trees. Furthermore, the API can transparently fall back to a copying approach when shared memory is not available, which allows the programmer to use it regardless of whether the parallelization is realized with local or remote processors.

The second approach exploits the fact that immutable data structures are inherently resilient against data races. Chapter 8 offers a definition of immutable classes together with a formal model for proving these correct. By exploiting the information hiding techniques offered by Eiffel, the language used by the principal implementations of Scoop and D-Scoop, the proofs can be mechanically generated without annotations going further than marking the classes that should be immutable. While the main purpose of immutable classes in this dissertation is to offer a way to limit memory use and copying on shared memory machines, it also has some advantages in a distributed context. Most importantly, immutable classes are also a useful tool to structure programs since they offer clients a very useful guarantee. The lack of annotations is the main advantage of this approach to other approaches for immutable classes.

By showing two approaches to integrate shared memory in an otherwise no-share model, we show that D-Scoop can indeed fill the need of a model that does not forsake shared memory in order to offer safety from data races.

There are still some specialized areas where a general model has a hard time to compete.

GPGPU General Purpose usage of Graphic Processing Units has become an important area for massive parallel computation. Due to their inherently different architecture, they cannot be treated the same as normal processors. Ongoing work by Kolesnichenko [33] is promising to fill this particular gap.

Parallelism. While our work takes advantage of shared memory, other frameworks geared towards efficient parallel computing provide APIs optimized for parallel algo-

rithms. *Slicing* is only providing a fundamental support for distributing this data, so there is a need for more advanced APIs building on top of it.

Distributed Parallel Computing. Another area is distributed parallelism. Clusters, nowadays often equipped with GPUs, are used for scientific and commercial computing for large problems. API's for that integrate well with the D-Scoop approach are an interesting pursuit, and similar to the work here, should unify GPU, local and distributed parallelism.

While it might very well be possible to integrate all these into the Scoop model, at some point it might not be the best way to go forward. A unified model has many advantages, but specialized models can be better at their specialty as they are not restricted by others. While concurrency and distribution have the advantage of being, to a certain degree, tolerant in respect of efficiency, high-performance computing is not. It is an area where every small increase in efficiency can save a lot of money, so sacrificing even a little bit of efficiency in order to get a nicer and more integrated model is difficult to argue for.

In general, developer time is worth more than execution time, so a safe concurrency and distribution model with reasonable support for parallelism, as shown in this thesis, is often the right way to go.

Bibliography

[1] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, September 1989.

[3] John K Bennett. *The design and implementation of distributed Smalltalk*, volume 22. ACM, 1987.

[4] Bertrand Meyer. SCOOP tutorial. `http://se.ethz.ch/~meyer/down/scoop/scoop_tutorial.pdf`, 2013.

[5] Andrew Birrell et al. Distributed garbage collection for network objects. Technical report, Systems Research Center, 1993.

[6] Andrew Birrell, Greg Nelson, Susan S. Owicki, and Edward Wobber. Network objects. In *Proc. SOSP 1993*, pages 217–230. ACM, 1993.

[7] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. *Object structure in the Emerald system*, volume 21. ACM, 1986.

[8] Guy E. Blelloch. NESL: A nested data-parallel language. Technical report, Carnegie Mellon University, 1993.

[9] Michael Butler, Tony Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *Communicating Sequential Processes. The First 25 Years*, pages 133–150. Springer, 2005.

[10] Denis Caromel, Ludovic Henrio, and Bernard P. Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459–495, 2009.

[11] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007.

[12] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[13] Claudio Corrodi, Alexander Heußner, and Christopher M. Poskitt. A graph-based semantics workbench for concurrent asynchronous programs. In *Proc. FASE 2016*, volume 9633 of *LNCS*, pages 31–48. Springer, 2016.

[14] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Communications of the ACM*, 14(10):667–668, October 1971.

[15] M. Crispin. RFC2060 - Internet Message Access Protocol. URL: `https://tools.ietf.org/html/rfc2060`, dec 1996. accessed: 2016-10-13.

[16] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computer Science & Engineering*, 5(1):46–55, 1998.

[17] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in AmbientTalk. In *Proc. ECOOP 2006*, volume 4067 of *LNCS*, pages 230–254. Springer, 2006.

[18] Distributed SCOOP website. `http://cme.ethz.ch/scoop/dscoop/`.

[19] Norihisa Doi, Yasushi Kodama, and Ken Hirose. An implementation of an operating system kernel using concurrent object oriented language abcl/c+. In *European Conference on Object-Oriented Programming*, pages 250–266. Springer, 1988.

[20] Eiffel Documentation: Concurrent Eiffel with SCOOP. `https://www.eiffel.org/doc/solutions/Concurrent%20programming%20with%20SCOOP`. Acc.: Apr. 2016.

[21] Eiffel Software. `http://www.eiffel.com/`, 2016.

[22] Grand Central Dispatch (GCD) Reference. `https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html`. Acc.: Apr. 2016.

[23] Go programming language. `http://golang.org/`, 2016.

[24] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a java-like language. In *Proceedings of the 16th European Symposium on Programming*, ESOP'07, pages 347–362, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] Michi Henning. The rise and fall of corba. *Queue*, 4(5):28–34, 2006.

[26] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-threaded active objects. In *International Conference on Coordination Languages and Models*, pages 90–104. Springer, 2013.

[27] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA 1993*, pages 289–300. ACM, 1993.

[28] Mark Hills, Traian Florin Şerbănuţă, and Grigore Roşu. A rewrite framework for language definitions and for generation of efficient interpreters. In *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA'06)*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 215–231. Elsevier Science, July 2007. also appeared as Technical Report UIUCDCS-R-2005-2667, December 2005.

[29] C. A. R Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17:549–557, 1974.

[30] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM (CACM)*, 21(8):666–677, 1978.

[31] E. B. Johnsen, O. Owe, and I. Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.

[32] Einar Broch Johnsen, Jasmin Christian Blanchette, Marcel Kyas, and Olaf Owe. Intra-object versus inter-object: Concurrency and reasoning in Creol. In *Proc. TTSS 2008*, volume 243 of *ENTCS*, pages 89–103, 2009.

[33] Alexey Kolesnichenko, Christopher M Poskitt, and Sebastian Nanz. Safegpu: Contract-and library-based gpgpu for object-oriented languages. *Computer Languages, Systems & Structures*, 2016.

[34] R. Greg Lavender and Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 483–499. Addison-Wesley, 1996.

[35] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[36] K. Rustan Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with frozen objects. In *Proceedings of the 2Nd International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 192–208, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Barbara Liskov. Distributed programming in argus. *Communications of the ACM*, 31(3):300–312, 1988.

[38] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM (CACM)*, 31(3):300–312, 1988.

[39] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *ACM Sigplan Notices*, volume 9, pages 50–59. ACM, 1974.

[40] Dominic Meier. Parallelism visualizer for scoop. Master's thesis, ETH-Zürich, 2014.

[41] Bertrand Meyer. Iso/ecma eiffel standard (standard ecma-367: Eiffel: Analysis, design and programming language), june 2006.

[42] Bertrand Meyer. Systematic concurrent object-oriented programming. *COMMUNICATIONS OF THE ACM*, 36(9):56–80, 1993.

[43] Bertrand Meyer, Alexander Kogtenkov, and Emmanuel Stapf. Avoid a void: The eradication of null dereferencing. In *Reflections on the Work of CAR Hoare*, pages 189–211. Springer, 2010.

[44] Mark S Miller, E Dean Tribble, and Jonathan Shapiro. Concurrency among strangers. In *International Symposium on Trustworthy Global Computing*, pages 195–229. Springer, 2005.

[45] B. Morandi, S. Nanz, and B. Meyer. Who is accountable for asynchronous exceptions? In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 462–471, Dec 2012.

[46] Benjamin Morandi, Sebastian Nanz, and Bertrand Meyer. Safe and efficient data sharing for message-passing concurrency. In *Proc. COORDINATION 2014*, volume 8459 of *LNCS*, pages 99–114. Springer, 2014.

[47] Benjamin Morandi, Mischael Schill, Sebastian Nanz, and Bertrand Meyer. Prototyping a concurrency model. In *Proc. ACSD 2013*, pages 170–179. IEEE, 2013.

[48] Sebastian Nanz, Faraz Torshizi, Michela Pedroni, and Bertrand Meyer. Design of an empirical study for comparing the usability of concurrent programming languages. In *Proc. ESEM 2011*, pages 325–334. IEEE Computer Society, 2011.

[49] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming*. PhD thesis, ETH Zürich, 2007.

[50] Piotr Nienaltowski, Volkan Arslan, and Bertrand Meyer. Scoop: Concurrent programming made easy.

[51] Object management Group, Inc. *Fault Tolerant CORBA*, version 1.0 edition, May 2010.

[52] Object management Group, Inc. *Common Object Request Broker Architecture (CORBA) Specification*, version 3.3 edition, November 2012.

[53] Object management Group, Inc. *Asynchronous Method Invocation for CORBA Component Model*, version 1.1 edition, August 2015.

[54] Semih Okur and Danny Dig. How do developers use parallel libraries? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE'12, pages 54:1–54:11. ACM, 2012.

[55] Perl programming language. `http://www.perl.org/`, 2016.

[56] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[57] James Reinders. *Intel threading building blocks – outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[58] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[59] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP 2010*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.

[60] Mischael Schill, Sebastian Nanz, and Bertrand Meyer. Handling parallelism in a concurrency model. In JoãoM. Lourenço and Eitan Farchi, editors, *Multicore Software Engineering, Performance, and Tools*, volume 8063 of *Lecture Notes in Computer Science*, pages 37–48. Springer Berlin Heidelberg, 2013.

[61] Mischael Schill, Christopher M Poskitt, and Bertrand Meyer. An interference-free programming model for network objects. In *International Conference on Coordination Languages and Models*, pages 227–244. Springer, 2016.

[62] Roman Schmocker and Alexey Kolesnichenko. Concurrency patterns in scoop, 2014.

[63] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[64] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 211–230, New York, NY, USA, 2005. ACM.

[65] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC'07. XXVI International Conference of the*, pages 3–12. IEEE, 2007.

[66] Scott West, Sebastian Nanz, and Bertrand Meyer. Efficient and reasonable object-oriented concurrency. In *Proc. ESEC/FSE 2015*, pages 734–744. ACM, 2015.

[67] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. *Object-oriented concurrent programming ABCL/1*, volume 21. ACM, 1986.

[68] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kie, un, and Michael D. Ernst. Object and reference immutability using java generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 75–84, New York, NY, USA, 2007. ACM.

List of Tables

List of Figures

List of Listings