Language Design for Atomicity, Declarative Synchronization, and Dynamic Update in Communicating Systems

Paweł T. Wojciechowski
`ptw [at] cs.put.poznan.pl`

November 26, 2007

Contents

Preface

In this book, we design novel language constructs and runtime algorithms for atomicity, declarative synchronization, and dynamic protocol update. They can be used to build communicating systems from modular protocols, that can be replaced dynamically.

- *Atomicity* provides guarantees that a set of operations executed in a network site (machine) can be regarded as a single unit of computation, regardless of any other operations occurring concurrently.
- *Declarative synchronization* is the mean to implement control of various concurrent actions or events in the system, by defining a synchronization policy (such as atomicity). The policy is defined using a set of rules, separately from the code of components. This approach allows protocol components to be reusable in different protocol stacks, and facilitates dynamic replacement of protocol components.
- *Dynamic protocol update* means transparent replacement of protocols at runtime, so that the use of services implemented by these protocols is not disrupted. Concurrent, dynamic replacement of protocol components located on different network sites occurs under control of *switching algorithms*.

The book has the following structure. We begin by discussing motivations and contributions. Then, we describe the versioning algorithms for concurrency control in atomic tasks. In the following chapter, we design the calculus of atomic tasks, i.e. atomic, roll-back free transactions that may have I/O effects. The calculus has a type system for static verification of data required by dynamic versioning, which guarantees that the constructs of the calculus are used correctly. Then, we describe two different approaches to declarative synchronization: (1) the calculus of concurrency combinators, with type-based verification of combinator satisfiability (which guarantees that the combinators are used correctly), and (2) a constraint language for the role-based synchronization. Next, we describe a model of dynamic protocol update, and give two example switching algorithms. Finally, we design the class-based object calculus of dynamic rebinding, and use it to show the application of atomic tasks and combinators when rebinding concurrent objects. In the appendix, we have included proofs of type soundness for the calculus of atomic tasks, including the proof of dynamic correctness of an example versioning algorithm.

* * *

I would like to thank prof. Jerzy Brzeziński for his support and encouragement. I also appreciated referee reports from reviewers who studied a draft of the book while it was being considered for publication.

A large part of the work described here was done while I was a postdoctoral fellow at Ecole Polytechnique Fédérale de Lausanne (EPFL). I would like to thank prof. André Schiper, head of the Distributed Systems Laboratory, for giving me a lot of freedom in defining the scope of my research, and for many useful remarks. I thank André, prof. Uwe Nestmann, Sergio Mena, Olivier Rütti, Daniel Bünzli, and Rachele Fuzzati, for many interesting discussions within our joint project: "Semantics-guided design and implementation of group communication middleware".

I also thank prof. Martin Odersky for giving me the opportunity to expand my knowledge of programming languages, through seminars and meetings.

I owe special thanks to Olivier Rütti for his work on the SAMOA protocol framework; the framework served as a testbed for several design ideas related to dynamic protocol update, and is a working proof-of-concept for adaptive group communication. The work on SAMOA will appear as part of Olivier's forthcoming PhD Thesis. I would also like to thank Vlad Tanasescu, who has worked in his semester project on the role-based synchronization in the OCaml programming language, for his valuable contribution.

My special thanks go to group members, in the alphabetical order: David Cavin, Richard Ekwall, Arnas Kupšys, Stefan Pleisch, Yoav Sasson, Péter Urbán, and Matthias Wiesmann, and to other colleagues—too many to mention all of them here—for making my stay at EPFL a pleasant and memorable experience. Last but not least, I would like to thank France Faille, the group's Secretary, for helping me with bureaucratic procedures.

Preparation of the book was supported in part by the State Committee for Scientific Research (KBN), Poland, under KBN grant 3 T11C 073 28. The opinions, findings, or conclusions expressed in these chapters are those of the author and do not necessarily reflect the views of the KBN.

Paweł Wojciechowski
August 2007

Chapter 1

Introduction

There is a growing interest in the design of new programming languages that combine features such as concurrency (or parallelism) and possibility of adding new code during program execution. These features support multi-core CPUs and dynamic software updating. Example applications are distributed services that must run ”non-stop”, such as financial trading, telephone switches, flight reservations, and air traffic control. Stopping ”non-stop” services results in loss of revenue; it may also compromise safety. The service providers must be therefore able to repair, update or extend their systems with minimal service interruption. Another class of distributed “non-stop” systems are various ubiquitous and pervasive devices. From pragmatic reasons, it should be possible to update or extend their software without human intervention. The common feature of all these systems is the ability to communicate. Communication enables useful applications but it also makes the implementation of dynamic update challenging. Dynamically updateable, distributed systems can be implemented using popular programming languages, that allow software components to be loaded and bound dynamically. However, in order to facilitate programming, and be able to guarantee robustness, novel programming abstractions are needed.

In this book, we have identified three essential programming abstractions: atomicity, synchronization, and dynamic update. They can be used to implement communicating systems from modular protocols, that can be replaced dynamically. *Atomicity* provides guarantees that a set of operations executed in a network site (machine) can be regarded as a single unit of computation, regardless of any other operations occurring concurrently; this property allows dynamic updates to be applied consistently. *Synchronization* is the mean to control various concurrent actions or events occurring on a single machine, as well as on distributed machines. To support dynamic update, it should be possible to rebind components at runtime, with guarantees of preserving any synchronization constraints between concurrent actions of the components. Finally, *dynamic update* means transparent replacement of distributed, communicating components on-the-fly, with some robustness guarantees; we call this operation *dynamic protocol update*. For instance, any software update of device A should not disturb another device that happens to communicate with device A at the same time. Traditional programming languages and middleware systems do not provide good support for implementing atomicity, synchronization and dynamic update in communicating systems, as we briefly explain below, and in the following sections.

Combination of two features: concurrency and irrevocable communication, makes the use of traditional atomicity constructs not suitable for implementing communicating systems. For instance, traditional monitors do not support internal concurrency, while local atomic transactions (as they are known in database systems) do not support operations with irrevocable effects (we will explain it in Section 1.2.1). Moreover, traditional synchronization constructs (such as those for atomicity) do not compose well. Thus, any system decomposition may require the programmer to analyze the code of individual components and, if required, to modify the synchronization code. Unfortunately, this procedure is cumbersome and error-prone. Also, it is difficult to implement systems, in which synchronization policy (such as atomicity) must dynamically change, based on the system's *modus operandi.* Therefore, some better language support of synchronization is required, which allows a synchronization policy to be expressed in a declarative way. Finally, concurrent, dynamic replacement of protocol components located on different network sites may often require to synchronize various actions in the system. This can be done using protocols that implement *switching algorithms.* The choice of a switching algorithm depends on the semantics of an application, which makes the design of updateable, distributed systems a difficult task. Therefore, some language and system support are needed, so that the algorithms could be applied transparently, based on the required properties of applications.

In this book, we describe the results of our work aimed at providing better language and system support for implementing dynamically updateable, communicating systems. We describe novel language constructs with runtime algorithms for atomicity, declarative synchronization, and dynamic update of protocols. Our language constructs are typed, with *type systems* [84, 85, 17] able to statically check some of the safety properties, which are critical for correct program behavior. They can make it easier to implement communicating systems with safety and robustness guarantees.

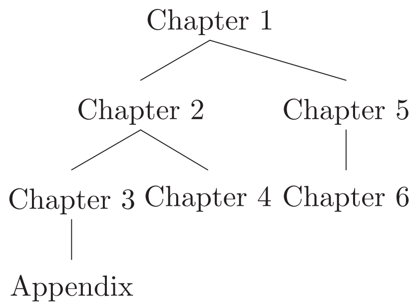


Figure 1.1: Chapter dependencies

Our book is aimed at researchers specialized in concurrent programming languages, distributed systems, and type theory. However, it may also be useful for readers from all areas of computer science who want an introduction to the rigorous design of concurrent programming languages. Our work builds on process calculi and operational semantics, which are the common means to introduce and define precisely any new language constructs and features. To avoid ambiguities of a pseudo-code, the operational semantics is also used to define selected concurrent and distributed algorithms.

The book assumes no preparation in the theory of programming languages, but readers should be familiar with at least one higher-order functional programming language (ML, Haskell, Scheme etc.), and with basic concepts of programming language design (abstract syntax, BNF grammars, evaluation, abstract machines, etc.). This material is available in many textbooks, for example *The Formal Semantics of Programming Languages: An Introduction* by Winskel [110], *Semantics of Programming Languages – Structures and Techniques* by Gunter [39], and in more recent books, such as *Types and Programming Languages* by Pierce [84] and *Practical Foundations for Programming Languages* by Harper [43]. Popular books on Standard ML include those by Paulson [82] and Ullman [104]. Some knowledge of distributed algorithms [65] will be also helpful.

The book has the following structure. In the remainder of this chapter, we define basic terms and discuss motivations and contributions. In Chapter 2, we describe the versioning algorithms for concurrency control in atomic tasks. In Chapter 3, we design the calculus of atomic tasks. In Chapter 4, we define two different approaches to declarative synchronization: the calculus of concurrency combinators, and the RBS constraint language for the role-based synchronization. In Chapter 5, we discuss the problem of dynamic protocol update, and present two example switching algorithms. In Chapter 6, we design the class-based object calculus of dynamic rebinding. In Chapter 7, we conclude. Finally, Appendix A includes proofs of type soundness for the calculus of atomic tasks, including the proof of dynamic correctness of an example versioning algorithm.

The major dependences between chapters are outlined in Figure 1.1. As the figure demonstrates, it is possible to start with Chapter 5 on dynamic update, following the introductory Chapter 1, and postpone Chapters 2, 3 and 4 on atomicity and synchronization. The Appendix may be left out by readers not interested in proof techniques.

1.1 Basic Terms

Let us begin from defining basic terms. A *protocol* defines an algorithm that solves a specific problem in a distributed system. For example, the TCP protocol solves the problem of having reliable, session-based point-to-point communication in an asynchronous network. Every protocol provides some service. Conversely, *service* is an interface (or specification) of all protocols that implement it. To work correctly, a protocol may also require some other services. For example, to provide the TCP service of reliable, session-based point-to-point communication in an asynchronous network, the TCP protocol requires the IP service. We will often use the notion "protocol" and "service" interchangeably, unless it is ambiguous.

Each protocol can be encoded by identical protocol modules, located on different network sites, where a *protocol module* (or *module* in short) is a software component that implements the protocol's interface (service). A *protocol stack* is a set of protocol modules that are composed together and located on a single network site. (We use the term "stack" due to historical reasons; actually protocol modules in a protocol stack can be composed in an arbitrary way, forming a graph.) Modules, possibly on different sites, can exchange *messages*. For example, the system in Figure 1.2 consists of two protocol stacks located on network sites s_1 and s_2; the stacks implement two protocols P and Q (composed into PQ) providing two services, respectively A and B. Services exchange messages (m_1, m_2, ..).

Note that the protocol composition defines different *levels of abstraction*. For instance, in the group communication middleware described in [70], an Atomic Broadcast (also known as Total Order Broadcast) is composed above a Distributed Consensus. The former service is used by the applications built on top of the middleware to deliver messages reliably and in the same order

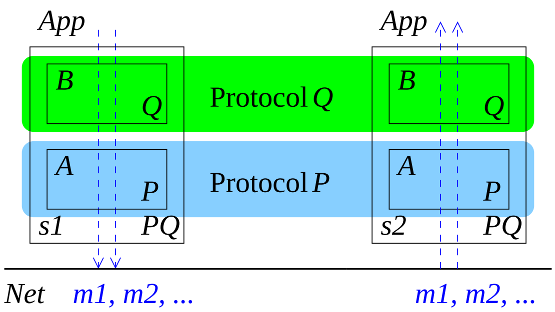


Figure 1.2: An example architecture of a communicating system

at all sites in the distributed system. The latter service (required by Atomic Broadcast) is used to deliver messages to every site despite of crash of some sites in the distributed system.

Protocol modules are bound or unbound to/from a service. At most one protocol module can be bound to a service at a time. When a service is called, the module bound to that service is executed. A module may get a message as the argument of the call; the message is processed by the module, and then it can be passed to another service, or to the application *App* on top of the stack. A computer network is represented as a service *Net*. Thus, by calling services, applications can exchange messages over the network.

The basic terms introduced in this section will suffice to understand material in this chapter; a more detailed model will be defined in Chapter 5.

1.2 Motivations and Contributions

Below we motivate our research, and give a capsule description of our contributions, which include: (1) the calculus of atomic tasks for atomicity, with type-based (static) verification of data required by dynamic versioning; (2) the calculus of concurrency combinators for declarative synchronization, with type-based verification of combinator satisfiability; and (3) a model of dynamic protocol update, with the class-based object calculus of dynamic rebinding.

1.2.1 Atomic tasks and versioning

It can be observed that due to the physical limit on clock speed, a growing number of desktop computers is being designed using new *multi-core CPUs*. Such CPUs have now started becoming available and will continue to materialise over the next several years. To fully exploit these future CPU through-put gains, applications (including protocols) will increasingly need to be multithreaded. Multithreading has already become an essential part of modern software systems. Although threads simplify the program's conceptual design and allow parallelism on multiple processors, they also increase programming complexity. Programmers must ensure that threads accessing shared data interact correctly, which is notoriously a difficult task.

Consider a protocol stack in which messages are processed by *concurrent threads* (this model of message processing appears in some protocol frameworks, e.g. the x-kernel [52, 123] and Cactus [121, 18]). Actions executed by a thread can be arbitrarily interleaved with actions of all other threads. We can therefore achieve a lot of parallelism on multicore CPUs. In practice, however, thread interleaving often must be constrained, e.g. to avoid interference between concurrent threads performing operations on shared data structures. Therefore, it must be possible to declare a set of protocol operations to be *atomic*. The execution of atomic operations appears as they would be executed alone, with no concurrency at all.

An *ad-hoc* implementation of atomicity, e.g. using fine-grain locks, is error-prone and may lead to deadlock. It is natural to ask whether *atomic transactions* could be used; they maintain the illusion of exclusive access to the whole data set while permitting concurrent access at a fine level. More precisely, the concurrent execution of transactions satisfies the *isolation property* [9], also known as *serializability* [108]. It ensures that operations of different transactions can be interleaved but the execution of transactions is equivalent (in terms of the transactions' effects) to some ideal serial execution of these transactions, in which these operations are not interleaved. Thus, the isolation property guarantees noninterference of transaction operations.

For example, an atomic transaction could be spawned upon receipt of every new message (of some kind) from the network. Then, the transactions would ensure noninterference for the message processing in the stack. Moreover, the order of delivering messages, e.g. to the application on top of the stack, will agree with the order of receiving messages from the network, which is often a required property. Contrary to single-threaded protocol stacks, however, this approach would allow protocols to better utilize multi-core CPUs, while at the same time it leverages the programmer from writing the low-level synchronization code (if the synchronization is required).

Example Let us consider an example system in Figure 1.2, which implements two services A and B on sites s_1 and s_2. The services are implemented by protocols, respectively P and Q. A protocol stack on site s_2 has received two messages: m_1 and m_2 (in the specified order); the messages are processed by service methods and delivered to the application *App* on top of the stack. We write, e.g., A^{m1}, to denote service A processing message m_1. For simplicity, we assume that each service method in the stack is atomic. Thus, concurrent processing of these two messages within our stack can be described by one of the following example traces of execution, or runs r:

1. $r_1 = A^{m1}, B^{m1}, A^{m2}, B^{m2}$ (a serial run)
2. $r_2 = A^{m1}, A^{m2}, B^{m1}, B^{m2}$ (an interleaved run)
3. $r_3 = A^{m1}, A^{m2}, B^{m2}, B^{m1}$ (a nested run)

Are all these executions *correct*? It depends on the specification of protocols. For instance, if services A and B would correspond to, respectively, the Distributed Consensus and Atomic Broadcast in the group communication stack (e.g. the one described in [70]), then the executions r_1 or r_2 on some sites cannot occur at the same time with the execution r_3 on other site(s). This is because the order of delivering broadcast messages to the application on top of the stack will not be the same on all sites, which contradicts the intended semantics of the Atomic Broadcast service.

However, if each incoming message would be processed by a fresh atomic transaction, then the isolation property of atomic transactions ensures that the execution r_3 cannot succeed, while executions r_1 and r_2 are permitted. Note that in r_1 and r_2, the transaction processing message m_2 can see the effects of another transaction processing message m_1, while the opposite is not true. On the other hand, in the execution r_3 (that does not satisfy the isolation property) the transaction processing m_2 can see an effect of the transaction processing m_1 through common service A, but the opposite also holds, i.e. the transaction processing m_1 can see the effects of processing m_2 through common service B.

The usual implementation of atomic transactions depends on *rollback-recovery*, as follows. Transactions are executed concurrently by the transaction manager. If some operations performed by transactions conflict with respect to isolation, i.e. the property cannot be guaranteed any longer, then the transactions are rolled back (restarted) and their state (if any) is recovered (recreated) from the transaction log.

For example, consider the nested trace $A^{m1}, A^{m2}, B^{m2}, B^{m1}$. If an atomic transaction processing message m_1 would intend to call service B, then this operation (B^{m1}) would be detected as conflicting with respect to isolation, and so it would cause all active transactions to rollback their execution. Thus, the protocol execution that is allowed to eventually complete, would be either r_1 or r_2, which is what we wanted. Unfortunately, it may not be possible to use the rollback-recovery approach in the implementation of protocol stacks, as we explain below.

The key issue in the design of support for atomicity in protocol stacks is the answer to the question: what is an *observational effect* of a protocol execution? For example, the observational effects in database systems are any changes of data stored in the database. In communicating systems, however, we also have *Input/Output (I/O)* effects, such as an input or an output of a message. Recovering I/O effects is problematic. For instance, re-sending messages that have already been output to the network may confuse remote participants of the protocol. It would be inelegant and inefficient to require protocols to deal with such erroneous cases. Thus, our goal was to design support of atomicity in communicating systems, which allows atomic code to have arbitrary I/O effects of its execution.

Do we need an *explicit* construct for transaction rollback? In database systems it can be useful, e.g. when implementing an interface with the database. A database client should be able to either “commit” changes made to the database, or “abort” these changes; the latter operation needs rollback. However, there is no such need in our case, since atomicity in protocol stacks is at the lower level of abstraction; if rollback would be required by some protocol, e.g. in order to make it fault-tolerant, then it had to be implemented as *part* of the protocol itself.

Based on the above observation, we decided to design support of atomicity (or isolation) that does not depend on rollback-recovery. The immediate advantage of this design decision was that the problem of irrevocable effects has been solved. The rollback-free execution of transactions can be achieved, e.g. by scheduling operations with observational effects, so that any conflicts between concurrent transactions can never occur.

Contribution

Our contribution in this area is the design of language support for atomic, rollback-free transactions, called *atomic tasks*. Atomic tasks ensure atomicity (isolation) without compromising task operations, i.e. it is plausible for atomic tasks to have I/O effects. Since tasks can be internally multithreaded, they provide a lot of flexibility in expressing atomicity in concurrent protocol stacks.

In Chapter 3, we describe an example design of atomic tasks. The implementation of atomic tasks can use our novel concurrency control algorithms that are based on *versioning* [119]; they schedule critical operations of concurrent tasks according to versions, so that the isolation property is preserved. In Chapter 2, we describe the idea of versioning and give example algorithms. The versioning algorithms, however, require some static (compile-time) data to work correctly. Depending on the variant of the versioning algorithm, these data can range from just names of *verlocks* [114] (or *versioning locks*) used to mark operations whose effects must be isolated, to some quantitative and spatial information; in short, the more data we can provide, the better interleaving of concurrent tasks we can achieve (i.e. more concurrency possible).

The versioning algorithms can no longer work properly, and atomicity is violated, if the data given to the versioning algorithms are wrong. We have therefore proposed in [114] that these data could be specified (or derived) as typing annotations of the language constructs that are used to spawn tasks, with the compiler able to verify if the typing annotations are given as required by the concurrency control algorithm. In practice, a positive round of a *type checker* will provide an automatic proof that our safety property, isolation, indeed holds in the protocol stack. Since the proof would be done at compile time, before a protocol is ever executed, it also makes obsolete any runtime checks and exceptions to handle any fatal errors or misbehaviour due to feeding the algorithms with incorrect data.

To formally show safety of our language, we have designed the *iso*-calculus of atomic tasks [114, 113], with a type system that can verify input data for an example versioning algorithm. The calculus and the algorithm are defined in Chapter 3. All expressions in the calculus whose execution might not guarantee isolation are rejected as "not well typed". The type system essentially guarantees two properties: (1) all operations that need to be isolated are protected by verlocks (*"no race conditions can happen"*)—this result builds on the work of Flanagan and Abadi [31] on safe locking (using standard locks) that we have adopted to our language of atomic tasks; and (2) all verlocks required by an atomic task are known on task creation (*"versioning is safe"*)—this result stems from our language.

The *iso*-calculus is equipped with operational semantics, that we have used to prove the *type soundness* theorem. The theorem states that the execution of any well-typed expression in our calculus cannot violate the isolation property (informally, "well typed programs can't go wrong"). The proof of the type soundness theorem for our calculus (in the Appendix) has three main parts: (1) type preservation, (2) progress, and (3) dynamic correctness of the versioning algorithm. The proof is rather lengthy and standard, except for part (3) that is nonstandard and can be instructive, e.g. it shows formal reasoning about concurrent algorithms in the style of operational semantics.

1.2.2 Declarative synchronization

One of the promises of modular protocol design is the possibility of reusing components implementing individual protocols (or services), to compose different protocol stacks. In practice, however, reuse of such code is problematic.

Consider protocol stacks that are built from multithreaded protocols. For instance, they could be built using *protocol frameworks* [116]—programming packages that allow complex protocols to be implemented as a collection of communicating components, each one implementing some sub-protocol (see, e.g. [52, 121, 76, 119, 92] for example protocol frameworks). These tools can help to support code reuse but they do not solve all the problems. For example, protocol stacks usually implement some *synchronization policy*, that is required to restrict unconstrained concurrency within the stack (we have seen an example of such requirement in the previous section). Reusing components of such stacks in another protocol stack is not straightforward; it may require to modify the code of components. In particular, any synchronization policy may need to be revised, which means removing or adding some synchronization constructs in the component code. This is however a counterexample to the unanticipated reuse of protocol modules as "black-boxes", since protocols must be reimplemented when they are reused. To solve this dilemma, we need programming abstractions for expressing any synchronization code independently from software modules (or components), and to correctly apply this synchronization when the modules are composed or executed. Our goal was to design such programming abstractions.

Contribution

Our contribution in this area are two different models and languages for *declarative synchronization*, which can be explained as follows. Programs are composed from components using standard language facilities, such as classes or modules. However, the error-prone synchronization code is no longer entangled in the code of components but expressed as synchronization constraints, using a specialized declarative language. This approach supports code reuse, and also greatly simplifies programming of complex synchronization policies.

In Chapter 4, we describe two approaches to declarative synchronization. In the first approach, an arbitrary synchronization policy can be declared in the form of a type expression that abstracts away from any details of the synchronization code. The type expressions are built (or composed) using *concurrency combinators*, which are higher-order functions, or operators for specifying: true parallelism, causal order, and atomicity; we denote them, respectively $\|$, $\triangleright$, and `isol` (and `foll` for a variant of `isol`). Atomicity provides guarantees that a set of local operations on a site can be regarded as a single unit of computation, regardless of any other operations that may occur concurrently.

In our original proposal, the concurrency combinators take as arguments names that can be bound to arbitrary code fragments. Extending this to an object-oriented language, such as Java [5], the concurrency combinators may take as arguments interface names, e.g. $A.m$, of methods or object fields, where A is some service, and m is a method or field of this service. By combining operators, complex policies can be declared.

Example Let us consider again our small example protocol stack in Figure 1.2, which implements a composite service AB consisting of two services A and B. We assume that protocols P and Q providing these services are synchronization-free, i.e. they can be composed in different protocol stacks, without worrying about any synchronization constraints. Then, we can use the concurrency combinators to specify several different synchronization policies SP within our stack, e.g.:

1. $\mathrm{SP}_1 = \{A.m \parallel B.n\}$
2. $\mathrm{SP}_2 = \{A.m \,\triangleright\, B.n\}$
3. $\mathrm{SP}_3 = \{z = A.m \,\triangleright\, B.n,\ \ z\ \mathtt{isol}\ z\}$

Policy SP_1 requires methods m and n of services A and B to be called by separate threads (i.e. each message in the stack can be processed by these services concurrently). Policy SP_2 requires that method $A.m$ should be called before method $B.n$ (i.e. methods m and n cannot be called in the opposite order by any composite protocol PQ). The last policy SP_3, in addition, also requires that any concurrent executions of PQ must guarantee atomicity (i.e. all network messages will be delivered to the application *App* in the same order as processed by *both* P and Q). The last property is required if, e.g. PQ would be a TCP/IP, an atomic broadcast, or another protocol that guarantees the order of message delivery.

Although, the language of concurrency combinators is fairly simple, it has introduced a new problem of *combinator satisfiability*: the programmer has to make sure that declarations of the synchronization policy expressed using our language of concurrency combinators, and the code of the main (reusable) protocol parts are compatible (e.g. $\parallel$ requires a new thread to be spawned).

In practice, checking this guarantee by hand or at runtime would be problematic and error-prone. We have therefore proposed that a type system could verify *statically* if this condition holds, thus giving an automatic proof that the program is correct with respect to combinator satisfiability. We stated our result formally by defining the CK-calculus of concurrency combinators, with a type system able to verify if programs expressed in our language satisfy a synchronization policy declared using the combinators.

We have also designed a different approach to declarative synchronization [102], in which arbitrarily complex synchronization policies can be defined as constraints between application-dependent *semantic roles*, such as producers and consumers; the constraints can be expressed using a constraint language. We use the term *role* to mean one or possibly many concurrent threads, spawned to execute some fragments of code; the threads logically represent the semantic roles. Roles are subject of some predefined, role dependent, synchronization rules (or "synchronization aspects"), which are predefined for a particular "role family". For example, the Producer-Consumer family has a rule which says that the Consumer should be blocked and wait till the Producer produces some value. The actual implementation of synchronization rules is provided separately by a corresponding *synchronization package*, implemented in the host language. In Chapter4, we describe a general idea of the role-based synchronization; the implementation details are in [102].

1.2.3 Dynamic protocol update and rebinding

The essence of dynamically updateable communicating systems is the ability to modify its own code at runtime, i.e. without stopping the system. There have been a lot of work on *dynamic software updating (DSU)* (see, e.g. [47, 28, 11, 4, 107, 66, 101, 12, 15, 105, 21, 57, 100]), and many implementations have been developed.

Much of the recent work in this area concentrates on novel language support for DSU that does not break abstraction and can guarantee type safety despite code replacement occurring at runtime (see, e.g. [66, 107, 47, 28, 12, 101]). Different solutions enable dynamic update of arbitrary code fragments, or only dynamic update of selected language structures. However, most of this research does not explicitly deal with problems of concurrency and communication, either assuming that updated code fragments are single-threaded and they do not communicate, or compromising safety guarantees.

On the other hand, many middleware and component systems have been developed, which support some form of dynamic code replacement (see, e.g. [105, 57, 21] and other references in Chapter 5). These systems use traditional programming languages, which allow whole objects or components to be replaced dynamically on a site. The problems of concurrency and communication are solved using some synchronization means, built into the runtime system. However, most of these implementations are rather *ad-hoc*, and often they have been developed for a particular application only.

In this book, we are especially interested in language and runtime support that is required to implement dynamic update of software systems, whose different parts are distributed, and may need to communicate over the network. We consider a problem of DSU, that can be defined by the following question: what needs to be done, if dynamic software update means the change of a network protocol? Or, in other words, how to dynamically replace protocol modules in protocol stacks, without interrupting services implemented by these stacks? We call this operation the *Dynamic Protocol Update (DPU).*

Consider the example protocol stack in Figure 1.2. We want to dynamically replace a distributed protocol P implementing service A, for some new protocol, also providing service A, so that a protocol Q (above in the protocol stack) does not even notice that the protocol replacement has occurred. To be able to do so, several local software updates or component replacements (each one occurring on a separate machine) must be somehow coordinated or synchronized. This is required in order to guarantee global service correctness and availability while all the local updates or replacements take place.

Although some *ad-hoc* implementations exist (see, e.g. [21, 105, 62, 10, 100] and other references in Chapter 5), they are not free from drawbacks, or they are specialized for switching of some kind of protocols only. It remains unclear what guarantees of dynamic protocol update are required in general, and how to best provide them in a distributed system. There is also little existing work on language support for DPU. Our goal was therefore to investigate language and system support required for DPU in the general case.

Contribution

In our model of DPU [118], network protocols implementing some service can be switched on-the-fly without interruption of the service. The protocol update occurs under control of a *switching algorithm* that, depending on a service, may synchronize all local updates of protocol stacks across distributed machines, or simply apply updated code lazily, without any global synchronization. Most of our work in this area is on the system side; it is therefore beyond the scope of this book, which is on the foundations and language design. Therefore, we have only described, in Chapter 5, our work on the formalization of basic DPU properties. We have used our model of DPU to specify and prove properties of two general-purpose switching protocols. They define design space frame for application-specific, efficient algorithms. For example, in [93, 91], we have described efficient DPU algorithms that we designed and implemented for group communication middleware.

Turning to the language design issues: what language features would be useful to implement modular, communicating systems with dynamic protocol update? The essential problem here is how to combine some form of *dynamic rebinding* (of objects or modules) with synchronization constructs. To address this issue, we have designed a concurrent, class-based object calculus of dynamic rebinding [115], working towards future dynamic object languages. Contribution of this work is largely preliminary. Our object calculus, described in Chapter 6, has been used mainly to identify basic properties of dynamic rebinding of objects to signatures (or object interfaces), and to show how the main concepts described in this book (such as atomic tasks and concurrency combinators) could be integrated with object-oriented features of programming languages.

Chapter 2

Versioning for Atomicity

The goal of this chapter is to present concurrent algorithms that can be used to implement atomicity in protocol stacks. In a protocol stack, some actions can be executed concurrently, in order to: 1) achieve good response time (for instance, when performing slow I/O operations), 2) enable simultaneous processing of different types of messages, and 3) gain benefits of the multi-processor architectures. However, in order to maintain certain consistency conditions, concurrent actions must often be synchronized according to some *synchronization policy*. Synchronization policies can be implemented using traditional language facilities such as locks, semaphores, and monitors. However, the synchronization code is rather subtle, error-prone, and does not compose well in modular protocol stacks (see Section 1.2.2).

A synchronization policy that is very useful in concurrent programming is *atomicity*. An atomic expression is executed in such a manner that all other threads of execution observe either that the computation has completed, or that it has not yet begun; no other thread observes an atomic expression to have only partially completed. In this book, we only consider atomic expressions that normally complete their execution (unless a process or machine has crashed). We also do not allow for rollback in protocol stacks (see Section 1.2.1 for the explanation why). We call such atomic expressions *atomic tasks*. By analogy to database systems, we can say that atomic tasks are like rollback-free *atomic transactions*, that only support *isolation* [9]. The isolation property ensures that the effects of one task are not visible to other tasks executing concurrently; from the perspective of a task, it appears that tasks execute sequentially rather than in parallel. In this chapter, we give a more precise definition of atomic tasks and isolation. In the next chapter, we use the operational semantics to define these two notions formally.

Unlike lexically-scoped atomic transactions whose scope is indicated using the “commit” and “abort” keywords, atomic tasks have scope defined by the task’s expression. For example, consider a protocol stack in which every message received from the network is processed by a fresh atomic task; the task comprises all service methods in the stack that must be executed in order to deliver the message to the application on top of the stack. Thus, the scope of such atomic tasks is dynamic, since different service methods may be called, depending on the actual stack composition and message semantics. An atomic task terminates when the task’s expression completes, and all threads spawned by the task have terminated.

The most important difference between atomic tasks and atomic transactions is, however, that the *effects* of task execution comprise both database effects (i.e. modifications to the data store) as well as the Input/Output (I/O) effects, such as an input (or an output) of a message. In order to easily support I/O effects, we decided that atomic tasks are never aborted. Another feature of atomic tasks is that they can be internally multithreaded.

In this chapter, we describe three concurrency control algorithms: the *Basic Versioning Algorithm (BVA)*, the *Supremum Versioning Algorithm (SVA)*, and the *Route Versioning Algorithm (RVA)*. They can be used to manage the execution of concurrent tasks in a protocol stack, so that the isolation property holds. The SVA and RVA algorithms are the more efficient variants of the BVA. They can support more parallelism by demanding some additional properties of protocols to be known in advance, i.e. before spawning a task. The SVA algorithm requires to specify for every task, the least upper bound (or supremum) on the number of times the protocol stack’s services can be called by the task. The RVA algorithm requires the pattern of service calls in the protocol stack to be known, i.e. information about which services may be called if some service has been called.

The algorithms have been first published in [119]. The presentation in this chapter, however, differs from the description in the paper. In [119], we have used an event model, in which protocol modules (or components) communicate in the protocol stack using events. In this chapter, on the other hand, we use a service model, which is closer to the semantics of the programming language designed in this book.

The chapter is organized as follows. Section 2.1 defines basic terms using a small example protocol stack. Section 2.2 explains the idea of task scheduling using versioning. Section 2.3 describes three example versioning algorithms. Section 2.4 briefly describes SAMOA—a protocol framework, which implements atomic tasks using the versioning algorithms described in this chapter. Finally, Section 2.5 discusses related work.

2.1 Model

We consider modular protocol stacks, in which protocol modules (or components) communicate by calling service methods. Below we define our model, and use it to illustrate the notion of the isolation property. In the next chapter, we define isolation formally, using the operational semantics of our language.

2.1.1 Services and protocols

Protocol modules are objects, each one implementing a single service (an interface of the object). Protocols are accessed by service methods, e.g. $A.m$ denotes a method m of a service (an interface) A. Executions of service methods are triggered by service calls, ranged over by α, β, γ. A *service call* is simply a request (at runtime) to execute a service method. Any service calls caused by the execution of a method triggered by a call α are *causally dependent* on α; the causality relation is reflexive and transitive. A call α is *pending* if a method requested by α has not commenced yet.

Service calls can be synchronous or asynchronous. *Synchronous calls* are characterized by the client invoking a service method and then waiting for a response to the request (i.e. a value returned by the method). With *asynchronous calls*, the client invokes a service method but does not wait for the response—it can continue with some other processing. Any value returned by the method is ignored; if any response is required, a specified callback should be made. To support concurrency in the protocol stack, service methods can be executed by concurrent threads of execution. However, we require that a service is a *critical section*, i.e. a protocol module bound to the service must not be concurrently accessed by more than one thread of execution.

All methods of a service are grouped into a single protocol module (object). Protocol modules are the smallest units of *protocol composition*. Execution of a service method can directly modify only a local state of its own protocol module. The state encompasses all of the in-memory and on-disk data items and also I/O effects that affect the protocol's operation, such as an input or an output of a message. Protocol modules can be composed to give a *composite protocol*. A local state of a composite protocol is the union of (disjoint) local states of all protocol modules composed into the composite protocol.

Execution of a single method can generate zero, one or more service calls. We consider two kinds of calls: internal and external. An *internal call* triggers the execution of another (or the same) service method in the same protocol stack. *External calls* are: 1) requests by one protocol stack to inject a message to another protocol stack, 2) requests by an application to inject a message to a

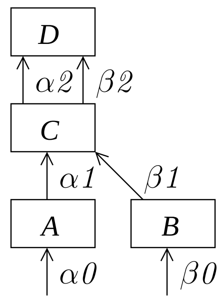


Figure 2.1: An example architecture of a modular system

protocol stack, and 3) requests to commence a new atomic task (we define the notions of a task and an atomic task below). We require that all external calls must be asynchronous. A sequence of external calls can be used to represent the I/O effects of a protocol.

Execution of a protocol is modelled as a *run*, defined as a list of pairs (α, A), where α is a service call and A is a service executed as the result of the call (for simplicity, we omit method names). If the execution of A has not commenced yet, we write `request` A. The list is ordered according to the time when the execution of the service commenced; in case of pairs $(\alpha, \texttt{request}\ A)$, where α is pending, the time of a call request is taken. A run is *complete* if it does not have pending calls.

Example protocol stack

To illustrate the notions introduced so far, let us consider an example protocol stack with four services A, B, C, and D, as illustrated in Figure 2.1. Our example stack can, for example, receive network packets either from an *ad-hoc* network (to be processed by A) or a fixed network (to be processed by B), and deliver them to D via the execution of C.

We assume that two external calls α_0 and β_0 have occurred. In all possible method executions triggered by these calls, the call α_0 (correspondingly β_0) triggers the execution of A (correspondingly B); and C is executed twice, once as the result of an internal call α_1 (which causally depends on call α_0), another time as the result of an internal call β_1 (which causally depends on call β_0). The calls α_0 and β_0 are concurrent, also the calls α_1 and β_1 are concurrent. Similarly, D is also executed twice, due to two concurrent calls α_2 and β_2.

Example complete executions (or runs) r_1, r_2 and r_3 in the protocol stack are:

$$\begin{aligned} r_1 &= ((\alpha_0, A), (\alpha_1, C), (\alpha_2, D), (\beta_0, B), (\beta_1, C), (\beta_2, D)) \\ r_2 &= ((\alpha_0, A), (\beta_0, B), (\alpha_1, C), (\alpha_2, D), (\beta_1, C), (\beta_2, D)) \\ r_3 &= ((\alpha_0, A), (\beta_0, B), (\alpha_1, C), (\beta_1, C), (\beta_2, D), (\alpha_2, D)) \end{aligned}$$

2.1.2 Tasks and atomic tasks

A *task* is a subsequence of a run, which begins from an external (asynchronous) call, say α, and consists of all pairs containing α and all calls that are causally dependent on α, but excluding any external calls β, γ, (...), that causally dependent on α, and any calls that may causally depend on β, γ, (...). The tasks spawned by external calls β, γ, (...) are *caused* by the task spawned by α. From the definition of an external call, we have that all tasks are spawned by asynchronous calls.

We require that all tasks must eventually complete. A task has *completed*, if the executions of all methods triggered by the task's service calls have completed, and no call is pending. In our example (see 2.1.1), we have two tasks: $k_\alpha = ((\alpha_0, A), (\alpha_1, C), (\alpha_2, D))$ and $k_\beta = ((\beta_0, B), (\beta_1, C), (\beta_2, D))$; they are not causally related because the calls α_0 and β_0 are not causally related. (We consider scope of a single protocol stack, not the whole distributed system, in which α_0 and β_0 *could* be causally related.)

Consider a complete run with a finite set of external calls $E = \{\alpha_0, \beta_0, ...\}$. The protocol execution (or a run) is *serial* if for each two (distinct) external calls α_0 and β_0 in E, either (each method of) the task spawned by α_0 commences after the task spawned by β_0 has completed, or vice versa. Moreover, in a serial run, a given task k always precedes in time any tasks caused by k, i.e. they can commence only after k has completed. In our example protocol stack, run r_1 is serial since task k_β begins in this run after k_α has completed; but runs r_2 and r_3 are not serial.

Two protocol executions (or runs) are *equivalent* if, considering the same sequence of external calls and the same initial state of the protocol, they produce the same state (or equivalent state, assuming some other notion of equivalence instead of equality). Then, we say that a protocol execution satisfies the *isolation property*, if the execution is equivalent to some serial execution (or a run) of the protocol. *Atomic tasks* are the tasks of a protocol execution (or a run) that satisfies the isolation property.

Consider runs r_1, r_2, and r_3 (within which we have the sequence of external calls α_0, β_0). Note that in runs r_1 and r_2, the task k_α visits all protocol modules that are shared with task k_β *before* k_β visits these protocol modules (recall that services are assumed to be critical sections). Thus, all memory and I/O effects of task k_β that change the state of the protocol composed from C and D, do not affect task k_α executed concurrently (since k_α cannot observe any of such changes to the state). This is exactly what occurs in a serial run. Hence, by definition runs r_1 and r_2 satisfy the isolation property.

However, run r_3 does not satisfy this property, since it is not equivalent to any serial run. To prove it, note, e.g., that task k_α can see any modification of D's data done by call β_2 of task k_β, and k_β can see any modification of C's data done by k_α. Since it is not possible to have a serial run in which this would be possible, we can conclude that run r_3 does not satisfy the isolation property. We can also prove this result in a different way. Note that any potential I/O effect of task k_β caused by D in this run would *always* precede I/O effects caused by D in task k_α, while in the serial run r_1, it is exactly opposite. Thus, the sequences of I/O effects in r_1 and in r_3 are different. So, these runs are not equivalent. Since r_1 is the only possible serial run in which call α_0 precedes β_0, which is required by r_3, this ends the proof. $\square$

When would atomicity be required in practice? Let us assume that our example protocol stack implements group communication in a distributed system. As the result of calls α_1 and β_1, two network messages, respectively m_1 and m_2, are passed to CD. Note that runs r_1 and r_2 would deliver these messages to the application on top of CD in the same order: first m_1, then m_2, while r_3 delivers the messages in the opposite order. Thus, if CD would be a group communication service, such as the Atomic Broadcast [40, 95], that must deliver messages at all sites in the same order, then we must somehow forbid the case when on some sites there are runs r_1 and r_2, while on other sites there is r_3. An example solution is to use atomic tasks, which permit only runs that satisfy isolation, such as runs r_1 and r_2.

2.2 Scheduling of Task Operations

We use the term isolation to mean that any memory and I/O effects of one atomic task are not visible to other tasks executing concurrently; from the perspective of a task, it appears that tasks execute sequentially rather than in parallel. Thus, the simplest possible solution to implement atomic tasks would be to block spawning of a new task until any other tasks complete. It follows from the definition in Section 2.1 that the isolation property is satisfied since the task's execution *is* serial. However, our goal is to support concurrency in the protocol stack. Roughly, a greater degree of concurrency leads to higher

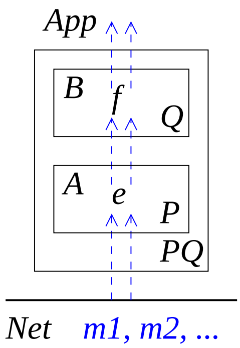


Figure 2.2: An example protocol stack, with critical operations e and f

performance on multicore CPUs. Essentially, we want the runtime system to process *many* atomic tasks simultaneously while providing the illusion of isolation; each new atomic task is therefore executed by a new thread. Moreover, we would like a task to be able to spawn another threads that belong to it. We then say that a task terminates if all its threads terminate.

Consider an example protocol stack in Figure 2.2, with two services A and B, which are implemented by protocols P and Q. Services A and B execute a critical operation, respectively e and f, where a *critical operation* is any operation that has a memory or an I/O effect, such as modification of a local state of a protocol module (object), or an input (or an output) of a message. We require that each network message m_1, m_2, (...) received by our stack is processed by a fresh atomic task.

Let us now explain the idea of using *versioning* for scheduling the execution of tasks, so that it satisfies the isolation property. In Figure 2.3, we illustrate two example atomic tasks: k_1 and k_2, processing messages, respectively m_1 and m_2. Figure 2.3(a) shows task scheduling using the BVA versioning algorithm (defined below). Note that a critical operation e of task k_2 is postponed until task k_1 has completed. This gives an almost serial execution of tasks. However, some other critical operation g of another service (not given in Figure 2.2) can be executed by task k_2 in parallel with task k_1; note that g is executed by a separate thread that belongs to task k_2. The isolation property is still satisfied since the effects of executing g cannot be observed by task k_1.

In order to postpone critical operations, the calls of services containing these operations can be temporarily *blocked*. This mechanism has been used in the implementation of atomic tasks in the SAMOA protocol framework (described later in this chapter). Another approach will be presented in the

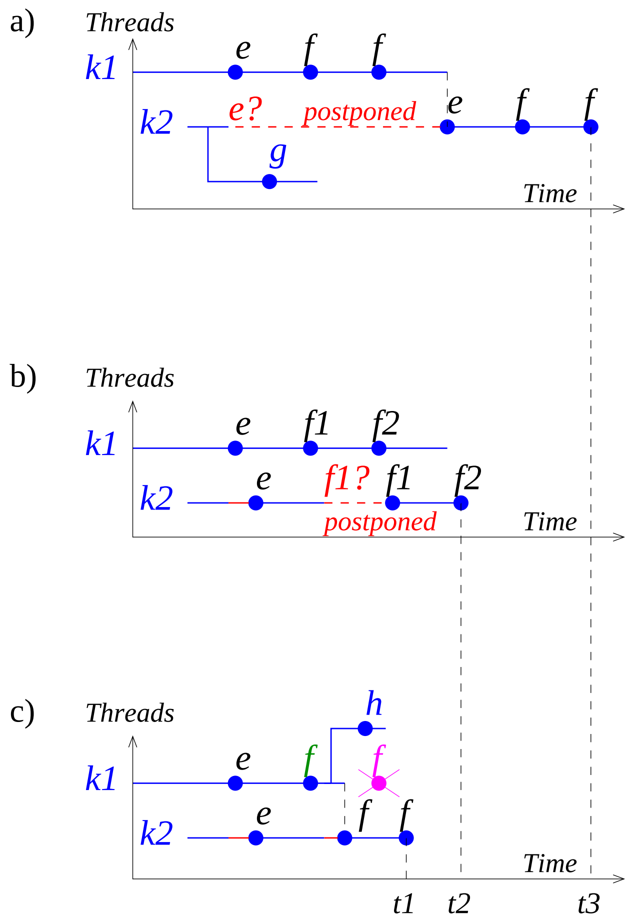


Figure 2.3: Task scheduling: a) BVA, b) SVA, and c) RVA

next chapter, where we design a language of atomic tasks. Our language is intended for general purpose concurrent programming, not only within the service model. It has a construct for guarding critical operations that are applied to an object, or applied to a communication channel of a given type. This mechanism allows for more fine-grain concurrency of task scheduling, and gives more flexibility since an arbitrary expression can be a critical section.

In order to schedule the execution of tasks, the versioning algorithms require some input data about a task to be known *a priori*—i.e. before the task is spawned. In case of the service model with call blocking, the BVA algorithm must know *a priori* for each new task, the names of all services whose methods may be called by the task. For instance, before spawning task k_1, the algorithm must know that k_1 may call services A and B, where "to call a service" means to call some method of this service. Similarly, before spawning task k_2, the algorithm must know that k_2 may call services A, B, and some other service (containing critical operation g).

Figure 2.3(b) shows task execution using the SVA algorithm. The algorithm must know *a priori*, a *least-upper-bound* (or *supremum*) on the number of times a given service may be called by an atomic task. For instance, service A executing operation e, is called by tasks k_1 and k_2 only once, while service B, executing operation f is called twice. This quantitative information allows the algorithm to permit more parallelism than BVA. For example, consider tasks k_1 and k_2. Note that the critical operation e of task k_2 (that had been postponed) is now executed soon after task k_1 has completed executing A (see Figure 2.3(b)). This is permitted since the SVA algorithm already knows that the supremum on the number of times service A (containing operation e) can be called by task k_1 is equal 1, i.e. the operation e will not be executed again by k_1, and so k_2 can call A and execute e. A similar situation occurs in the case of service B executing operation f, except that the supremum for B is now equal 2; this means that k_2 must be blocked on the call of B till k_1 has completed its second call of B, thus executing f for the second time. If supremum cannot be reached for some service, e.g. because it has been over-calculated, or because a branch of the program has been chosen that does not call this service, then the SVA algorithm performs like BVA.

The SVA algorithm could be improved, if we could somehow explore the spatial information about task execution. This is precisely what the RVA algorithm does. Figure 2.3(c) shows an example execution of two tasks, k_1 and k_2, under control of the RVA algorithm. For each task, the algorithm must know *a priori*, a causal relation between service calls made by the task. In our example, two such relations are possible for services A and B: either A calls B twice, or A calls B and B calls itself. Let us assume that the latter holds. Now, the key is behaviour of task k_1. Task k_1 has *diverged*, e.g. it has executed the 'else' clause instead of 'then', and, instead of calling itself and re-executing operation f, it has called another service C (not given in Figure 2.2) that executes some critical operation h. Task k_1 has not completed yet, but from the causality relation we know that C will never call B. Thus, after services A and B have completed, the RVA algorithm knows that task k_1 is not going to repeat operation f (of service B). It thus allows another task, k_2, to call service B and execute f for the first time. Note that in case of the BVA and SVA algorithms, the call of B by k_2 in the last scenario would be postponed till *all* services of k_1 complete (and k_1 would terminate).

If we denote by t_1, t_2 and t_3 the times of completing tasks k_1 and k_2 by the algorithms, respectively RVA, SVA, and BVA, then we have $t_1 \leq t_2 \leq t_3$ (see Figure 2.3). The SVA and RVA algorithms can be combined to obtain a more efficient algorithm, which permits even more parallelism then either of the two algorithms individually.

2.3 Versioning Algorithms for Atomic Tasks

In this section, we describe the BVA, SVA, and RVA concurrency control algorithms for atomic tasks. The key idea behind these algorithms is *versioning*: a service method called by a task is executed if the task holds a matching *version number* of the service. Otherwise, the call is blocked, waiting for the version upgrade. Thus, version numbers determine the order of executing service methods by atomic tasks; this order agrees with the isolation property. This simple mechanism protects protocol modules (or objects) from being accessed by atomic tasks that—in order to satisfy the isolation property—should wait till other atomic tasks access these modules (objects). We assume that they can be accessed by these tasks only through service calls.

We have used the versioning algorithms in the SAMOA protocol framework, described in the next section; they implement atomicity in concurrent protocol stacks. For simplicity, in the description of the algorithms, we assume that services are bound to protocols at startup and cannot be rebound. In SAMOA, however, this restriction does not hold—services can be dynamically rebound to new protocols providing the same service.

2.3.1 Basic versioning

Below we describe the *Basic Versioning Algorithm (BVA)* that can be used to implement a construct `atomic` $M\ e$. Execution of this construct spawns a new atomic task, where M is a set containing names of all services whose methods may be called by the task, and e is some expression that calls the first service (or services) of the task. For example, task k_α (see Section 2.1.2) could be spawned with `atomic` $\{A, C, D\}\ A.m\ v$, where A, C and D are the names of services, and an expression $A.m\ v$ is the call of a method $A.m$, passing v as the argument of the call.

Below are the rules of the BVA algorithm. For each service A in the protocol stack, there is a global version counter gv_A (one per stack), initialised to 0. Each individual service A maintains its local version counter lv_A, also initialised to 0. For brevity, we often omit the word "counter", simply saying "a version of a service" instead of "a version counter of a service".

Definition 1 (Basic Versioning Algorithm (BVA))**.** The algorithm is given by the following set of rules or steps:

1. At the moment of spawning a new atomic task k by `atomic` M e, for each service $A \in M$, increase the gv_A version counter of A by one.

   Create a private copy pv_k of all service versions computed as above, i.e., pv_k is a map (dictionary) containing bindings from all services $A \in M$ to copies of their upgraded versions gv_A.

2. A method of a service A called by a task k is executed only when the task holds a version for this service that matches the current (local) version maintained by the service, i.e.

   $$pv[A]_k - 1 = lv_A \quad . \tag{2.1}$$

   Otherwise, the call is pending.

3. After a task k has completed its execution, i.e. all its threads terminated, for each service $A \in M$ in parallel, wait until (2.1) is true, then upgrade the local version of A, so that we have $lv_A = pv[A]_k$; in the end, erase map pv_k.

We require Steps 1 and 2 to be critical sections (atomic).

We propose the following lemma:

Lemma 1 (Isolation Property by BVA)**.** Provided that service methods called by atomic tasks are executed only when allowed by the BVA versioning algorithm, and assuming that the algorithm has correct input data, the concurrent execution of the atomic tasks satisfies the isolation property.

In Chapter 3, we will define a similar versioning algorithm using an operational semantics of our language, and give a mathematical proof of its correctness. Below we only sketch an informal proof of the above lemma.

Proof (sketch) Consider a composite protocol with just two atomic tasks k_1 and k_2 that may call different methods of services declared in, correspondingly M_1 and M_2. Initially, all services have their local and global versions equal zero. The task spawned first (say k_1) will atomically increase the global versions of services in M_1 by one, and build its private set of versions (here, each version equal 1); see Rule 1. The task k_2, spawned after k_1, will also get its private set of versions for all services declared in M_2; however, versions of those services that have been also declared in M_1 will be equal 2 (again by Rule 1).

Consider a service A in $M_1 \cap M_2$. Each method of this service can be freely called by task k_1 since the task's private version of A decreased by one is 0, which is equal the current local version of the service, thus satisfying the equation (2.1) in Rule 2. However, by Rule 2 none method of A can be currently called by k_2 since k_2 holds a private version of A equal 2 and $2 - 1 = 1 \neq 0$. However, by Rule 3 the local version of A will equal 1, that is the private version of A hold by k_1, after k_1 terminates. Then, k_2 is allowed to call a method of A (by Rule 2). Since at this moment k_1 has already terminated, any changes done by k_2 to the state of A's object cannot affect k_1, which is what we wanted.

Consider a service A in $M_1 \cap M_2$ that has not been called yet by any of the two tasks k_1 and k_2. If task k_2, whose private version of A is newer than the private version of A hold by k_1, is about to terminate (according to our assumptions, it does not need to call a method of A to be allowed to terminate), it has to upgrade the local version of A. However, by the wait condition in Rule 3, it will be allowed to do so only *after* task k_1, which has an older version of A, will terminate. This mechanism prevents task k_2 from invalidating any version match of k_1, that entails k_1 to call A.

Essentially, by Rule 1 and the wait condition in Rule 3, we ensure that the order of upgrading, in Step 3, the local versions of services that are shared by concurrent tasks, is the same as the order of increasing global versions by these tasks in Step 1. This is precisely the necessary correctness condition for isolation provided by version-based concurrency control. The rest of the proof is by induction on tasks. □

We have also designed versioning algorithms that are variants (or optimizations) of the BVA algorithm. They permit to have more parallelism by upgrading local version counters as soon as possible. In the worst case, they behave like the BVA algorithm. However, they demand some additional (orthogonal) properties of protocols to be known before a task can be spawned. In the remainder of this chapter, we describe two such variants: SVA and RVA, which can be combined together.

2.3.2 Supremum versioning

Below we describe the *Supremum Versioning Algorithm (SVA)* that can be used to implement a construct `atomic` M *supremum* e. Execution of this construct spawns an atomic task, where M and e are as in Section 2.3.1, and *supremum* is a map (dictionary) from service names in M to the *least-upper-bound* (*supremum*) on the number of times the service can be visited by the task, where "to visit a service" means to call any service method of the service. We write $supremum[A]_k$ to denote task k's supremum for service A.

A supremum value allows the SVA algorithm to decide if a given service A that has been visited by a task k may be revisited by k, or not. If the latter, then the service's local version can be safely upgraded, permitting any other tasks to visit A. After supremum has been reached, any other task holding a matching private version, that wants to call any method of A, will be allowed to call it, and proceed concurrently with k, thereby enabling more parallelism than in the case of BVA, where k must firstly complete.

Giving the exact number of visits (which would be ideal) is not possible if a program branches. Since this is the case in almost any program, the algorithm only requires supremum. If a service is visited *less* times than declared in *supremum*, nothing wrong happens, just that less parallelism may be permitted compared with cases when *supremum* is more accurate. When it is not possible to predict the supremum number of visits, e.g. in programs that use recursion, a runtime error exception is thrown if supremum is exhausted; any compensation code can be then executed.

Below are the rules of the SVA algorithm. The algorithm is similar to the BVA algorithm; see Section 2.3.1 for the definition of gv and lv.

Definition 2 (Supremum Versioning Algorithm (SVA))**.** The algorithm is given by the following set of rules or steps:

1. At the moment of spawning a new atomic task k by `atomic` M *supremum* e, for each service $A \in M$, increase the gv_A version counter of A by $supremum[A]_k$, where $supremum[A]_k$ is the least upper bound of times the service A can be visited by task k.

   Create a private copy pv_k of all service versions computed as above, i.e., pv_k is a map (dictionary) containing bindings from all services $A \in M$ to copies of their upgraded versions gv_A.

2. A method of a service A called by a task k is executed only when the task holds a version for this service that matches the current (local) version maintained by the service, i.e.

$$pv[A]_k - supremum[A]_k \leq lv_A < pv[A]_k \quad . \tag{2.2}$$

   Otherwise, the call is pending.

3. Each time a service method $A.m$ called by some task k has completed, i.e. $A.m$ has returned and any threads of $A.m$ have terminated, the service's local counter lv_A is incremented by one.

4. After a task k has completed its execution, i.e. all its threads terminated, check if any local versions lv_A of services $A \in M$ need to be upgraded, i.e. there is $lv_A < pv[A]_k$; if so then for each such a service A in parallel, wait until (2.2) is true, and then upgrade the local version of A, so that we have $lv_A = pv[A]_k$; in the end, erase map pv_k.

We propose the following lemma:

Lemma 2 (Isolation Property by SVA)**.** Provided that service methods called by atomic tasks are executed only when allowed by the SVA versioning algorithm, and assuming that the algorithm has correct input data, the concurrent execution of the atomic tasks satisfies the isolation property.

Below we sketch a proof of the above lemma.

Proof (sketch) The proof of Lemma 2 is similar to the previous proof. The main difference is that task k_2 is allowed to call a method of service A soon after the local version lv_A of the service (initially equal 0) is equal $pv[A]_{k_1}$, that is either after k_1 has visited A the number of times declared in Rule 1 by $\mathit{supremum}[A]_{k_1}$, or, by Rule 4, after k_1 has terminated (in the case when k_1 visited A less times than declared). Note that if the least upper bound was too small, and k_1 would try to visit A more times than it has declared, then by Rules 1 and 3, lv_A will be equal or greater than $pv[A]_{k_1}$, and so by Rule 2, k_1 is not allowed to call any method of A since $lv_A \not< pv[A]_{k_1}$.

Rule 4, applied after k_1's termination ensures that any local version lv_A such that $lv_A > pv[A]_{k_1}$, i.e. upgraded by some other task(s) than k_1 after k_1's supremum had been reached, is never *downgraded* by Rule 4. Moreover, by Rule 1 and the wait condition in Rule 4, we have that the order of upgrading, in Step 4, the local versions of services that are shared by concurrent tasks, is the same as the order of increasing global versions by these tasks in Step 1. This is precisely the necessary correctness condition for isolation provided by versioning. The rest of the proof is by induction on tasks. □

2.3.3 Route versioning

Below we describe the *Route Versioning Algorithm (RVA)* that can be used to implement a construct `atomic` G e. Execution of this construct spawns an atomic task, where G is a directed graph defining a *routing pattern* of the task, and e is the code that calls the first service (or services) of the task.

Vertices of graph G are names of all services whose methods may be called by the task; each such service must appear in the graph exactly once. One vertex of G, named θ, is a "dummy" service that represents the expression e.

An arrow in graph G is a directed pair $A \to B$, for some services A and B, declaring that executing service A may result in calling service B. The arrows of the form $A \circlearrowleft$, for some service A, are also possible, declaring recursive calls of A. For instance, task k_α (see Section 2.1.2) could be spawned with `atomic` $\{\theta \to A \to C \to D\}$ $A.m\ v$, where A, C and D are the names of services, and an expression $A.m\ v$ is the call of a service method $A.m$, passing v as the argument of the call.

In SAMOA, a routing pattern G is built for each task dynamically at the stack composition time, based on static declarations; for each service in the stack, they specify which services are required (can be called) by the service. If service A of some task would try to call service B but there is no arrow from A to B in the graph G, then a runtime exception is thrown, and can be handled by some compensation code.

To gain benefits of the RVA algorithm, service calls must be asynchronous, i.e. after invoking a service method the client continues its execution; any value returned by the method will be ignored.

Below are the rules of the RVA algorithm; see Section 2.3.1 for the definition of gv and lv. We write $A \in G$ to mean that A is a vertex in graph G.

Definition 3 (Route Versioning Algorithm (RVA))**.** The algorithm is given by the following set of rules or steps:

1. At the moment of spawning a new atomic task k by `atomic` $G\ e$, for each service $A \in G$, increase the gv_A version counter of A by one, and create a private copy pv_k of all service versions computed as above, i.e. pv_k is a map (dictionary) containing bindings from all services $A \in G$ to copies of their upgraded versions gv_A.

   Create a private copy G_k of graph G, where each service (vertex) in graph G_k is paired with *status*, which is equal 0 for "inactive" services, and a number greater than 0 for "active" services. Initially, the service θ has status equal 1, and all other services have status equal 0.

2. A call of any method of a service A, made by a task k, increases the status of A in G_k by one.

   The called method will commence when the following two conditions hold (otherwise the call is pending):

   (a) task k holds a version for service A that matches the current (local) version maintained by A, i.e.

   $$pv[A]_k - 1 = lv_A \quad , \tag{2.3}$$

(b) there is in graph G_k, a directed arrow to A from the service that has made this call.

3. Each time a service $A \in G_k$, called by a task k, has completed its execution, decrease the status of A in graph G_k by one.

   If A's status is 0, then remove from G_k all "inactive" services x (i.e. with status $= 0$) that are not reachable from any "active" services (i.e. with status > 0), and for each such service x in parallel, wait till (2.3) is true, then upgrade a local version of x, so that $lv_x = pv[x]_k$.

   We say that service A is *not reachable* from service B in graph G_k, if there is no directed route from B to A in this graph.

4. After a task k has completed its execution, i.e. all its threads terminated, erase map pv_k.

We propose the following lemma:

Lemma 3 (Isolation Property by RVA)**.** Provided that service methods called by atomic tasks are executed only when allowed by the RVA versioning algorithm, and assuming that the algorithm has correct input data, the concurrent execution of the atomic tasks satisfies the isolation property.

Below we sketch a proof of the above lemma.

Proof (sketch) The proof of Lemma 3 is similar to the proof of Lemma 1. The main difference is that task k_2 is allowed to call a method of service A soon after the local version lv_A of the service is equal $pv[A]_{k_1}$, that is after A is removed from graph G_{k_1} by Rule 3.

Note that if the routing pattern G_k declared for k omitted some routes, then by Rule 2(b), k is not allowed to call any methods whose services are not reachable in G_k. However, there is no problem if the pattern provides routes to some methods that are never called by k, since by Rule 3, these methods will be eventually released (at the latest after k has completed).

The wait condition in Rule 3 ensures that local versions are never downgraded. Moreover, by Rule 1 and Rule 3, we have that the order of upgrading, in Step 3, the local versions of services that are shared by concurrent tasks, is the same as the order of increasing global versions by these tasks in Step 1. This is precisely the necessary correctness condition for isolation provided by versioning. The rest of the proof is by induction on tasks. $\square$

2.4 The SAMOA Protocol Framework

Below we briefly describe *SAMOA* [119, 92, 94]—a novel protocol framework in Java, that can be used for implementing protocols from communicating components. To our knowledge SAMOA was the first protocol framework that had built-in rollback-free, atomic tasks; the implementation of atomic tasks uses the versioning algorithms. Below we compare support of atomicity in our framework with two other similar programming tools: Cactus [121, 18], which builds on the x-kernel [52, 123], and Appia [76, 1].

Consider the example protocol stack in Section 2.1.1. If we would implement it using Cactus, then all given traces of execution (r_1, r_2, and r_3) are possible when processing concurrent messages. To forbid some traces, e.g. in order to provide atomicity, the programmer must write code synchronizing the execution of protocol methods (or event handlers, using the Cactus event model). In SAMOA, service methods triggered by an external call can be executed atomically if they are part of an atomic task, spawned using just a single construct. Then, only runs that satisfy the isolation property are permitted (e.g. r_1 and r_2, but not r_3). Appia also supports the isolation property, but it only permits serial executions of protocols, such as r_1. Other correct, concurrent runs, such as r_2, cannot occur in Appia. Below we briefly describe other features of the SAMOA protocol framework (see [92] for more details). More information about Cactus and Appia can be found in [116, 117].

A protocol module in SAMOA, implementing some service, consists of three functional parts: executers, listeners and interceptors. *Executers* handle two kinds of requests: a request to process a new message, and a request to send a message to the network. *Listeners* handle replies and notifications caused by the requests (usually made on another site), where a notification can be handled by listeners of many protocol modules, while a reply is handled by exactly one listener (of a single protocol module). An *interceptor* is a unique feature of the SAMOA framework; it allows to intercept requests, replies and notifications. Interceptors can be used to implement switching algorithms for dynamic protocol update (see Section 1.2.3), so that the execution of these algorithms is transparent to the protocols composed into the stack.

Protocol modules in SAMOA communicate through objects called service interfaces. Binding of service interfaces with protocol modules occurs according to certain principles. For instance, binding an executer to a service interface requires that a correct listener is bound to the service interface. This provides some guarantees that requests are correctly linked with replies or notifications. Executers, listeners and interceptors of a protocol module can be dynamically bound or unbound to a service interface. This mechanism allows protocols to be dynamically loaded to the protocol stack and replaced on-the-fly. Other features of the SAMOA protocol framework include message flow control, which removes possible bottlenecks in the protocol stack, and timeouts, which facilitate protocol implementation.

To validate our protocol framework, it has been used to implement an adaptive group communication middleware [93, 91], based on the modular approach to group communication [70]. Group communication is an important enabling technology for building fault-tolerant networked systems by server replication; it is therefore a good example of a distributed system that must run non-stop, with a high level of robustness. Some requests in the group communication stack can be executed concurrently, e.g. for better response time, or to avoid blocking of some actions. For this, the SAMOA atomic tasks have been used. The isolation property of atomic tasks ensures that every such request is processed by protocols in the stack, using a consistent set of data. Contrary to traditional synchronization constructs, atomic tasks made programming easier and less error-prone. Atomic tasks have also facilitated the implementation of dynamic protocol replacement; in Section 5.4, we provide some details on replacement of group communication protocols.

2.5 Related Work

The work in this chapter builds on research in the area of concurrency control algorithms for atomic transactions.

Research on transaction management began appearing in the early to mid 1970s. Quite a large number of concurrency control algorithms have been proposed for use in centralised and distributed database systems. Database systems use concurrency control to avoid interference between concurrent transactions, which can lead to an inconsistent database. Isolation is used as the definition of correctness for concurrency control algorithms in these systems. The algorithms generally fall into one of three basic classes: *locking* algorithms, *timestamp* algorithms, and *optimistic* (or certification) algorithms. A comprehensive study of example techniques with pointers to literature can be found in [9]. Concurrency control problems had been also treated in the context of operating systems beginning in the mid 1960s. Most textbooks on operating systems survey this work, see e.g. [99, 103].

Our versioning algorithms have some resemblance with basic two-phase locking. However, instead of acquiring all locks needed (in the 1st phase) and releasing them (in the 2nd phase), tasks take and dynamically upgrade version numbers, which optimizes unnecessary blocking. The conflicting operations are ordered according to version numbers, which is similar to ordering *timestamps* in timestamp algorithms [9, 108]. However, we associate versions ("timestamps") with services, not with transactions. Therefore all service calls are always made in the right order for the isolation property (the call requests with too high versions are simply delayed), unlike common timestamp algorithms for database atomic transactions, where if an operation has arrived too late (that is it arrives after the transaction scheduler has already output some conflicting operation), the transaction must abort and be rolled back. The "ultimate conservative" timestamp algorithms avoid aborting by scheduling all operations in timestamp order, however, they produce serial executions (except complex variants that use transaction classes) [9].

Methods of *deadlock avoidance* in allocating resources [99, 103] are also relevant to our work. The *banker's algorithm* (introduced by Dijkstra [25]) considers each request by a process as it occurs, and assigns the requested resource to the process only if there is a guarantee that this will leave the system in a safe state, that is no deadlock can occur. Otherwise the process must wait until some other process releases enough resources. The *resource-allocation graph* [49] algorithm makes allocation decisions using a directed graph that dynamically records claims, requests and allocations of resources by processes. The request can be granted only if the graph's transformation does not result in a cycle. Resources must be claimed a priori in these algorithms.

In our case, a task must also know a priori all its resources (methods or protocols) before it can commence. However, the history of service calls by different tasks is always acyclic since versions impose a total order on call requests performed by different tasks. Since tasks are assumed to complete, old versions will be eventually upgraded. Therefore our versioning algorithms are deadlock-free. Moreover, the calls are assigned according to the order that is necessary to satisfy the isolation property, unlike the resource allocation algorithms, which do not deal with ordering of operations on resources.

Chapter 3

Atomic Tasks

In this chapter, we describe a language of atomic tasks. The main construct of our language is `atomic` that spawns an atomic task. Tasks are analogous to multithreaded transactions decomposed to ensure an isolation property only. The isolation property of atomic tasks ensures that their concurrent execution is equivalent to an execution in which the tasks would be executed sequentially. Contrary to other similar language constructs, atomic tasks in our language can perform arbitrary operations, including operations with the I/O effects. This is achieved by tightly controlling the order of critical operations and guaranteeing that once started they cannot run into conflicts, i.e. rollback-recovery that is problematic for irrevocable I/O effects is avoided (see Section 1.2.1).

No explicit rollback construct is currently available in our language of atomic tasks. Since we intend our language for implementing protocol stacks, this is not a drawback. Consider a concurrent protocol stack that spawns an atomic task for communicating a message to atomic tasks on remote sites; the message is delivered by the tasks to an application built on top of the protocol stacks. Full-scale recovery of atomic tasks located on different sites would require some form of distributed agreement between sites. The distributed agreement algorithms, however, do not scale to large networks. Moreover, the application on top of the stack would also need to be able to rollback its state; requiring this for all applications is impractical.

Our language is defined formally as a small calculus of atomic tasks; the calculus is equipped with an operational semantics built on the standard call-by-value λ-calculus [87]; the semantics has been split into a dynamic semantics of the host language constructs, and of the concurrency controller. The key concept of our design is the use of a type system to support safe runtime execution of atomic tasks. We present a first-order type system that can verify input data for an example versioning algorithm that is used to implement the concurrency controller.

We have used the operational semantics to formalize and prove *type soundness* and the runtime correctness of the versioning algorithm. In particular, we show several results (theorems) about our type-directed approach to concurrency control of rollback-free atomic tasks. The main result is that the execution of any well-typed program in our language is guaranteed to satisfy the isolation property. In the Appendix, we give a rigorous proof of isolation preservation and progress (up to deadlocks between threads of the same task); the proof makes data and I/O accesses explicit, and deals with multiple threads within an atomic task. This was one of the first such proofs for atomic tasks. The results described in this chapter have been published in [114]; the proofs of type soundness appeared in [113].

The chapter is organized as follows. Section 3.1 motivates the design choices in our calculus of atomic tasks. Section 3.2 explains the constructs using an example program. Section 3.3 defines syntax, semantics, and typing of the calculus. Section 3.4 states the main results, including type soundness and dynamic correctness of the versioning algorithm, and Section 3.5 discusses related work.

3.1 Design Choices

3.1.1 Fine-grain, rollback-free concurrency control

The main construct of our language is $\texttt{atomic}\ \overline{x}\ e$, which can be used to spawn an atomic task, where $\overline{x}$ will be described below, and e is the task's expression. For rollback-free execution of atomic tasks, the language could be implemented using any of the versioning algorithms described in Chapter 2. In fact, we define an abstract machine of our language using a *Versioning Algorithm (VA)*, which is the BVA algorithm described in Chapter 2, slightly modified to work with a fine-grain synchronization construct (described below). The VA algorithm allows critical task operations to be scheduled according to the isolation property. It is one of the simplest algorithms possible, and so it does not permit much concurrency. However, it makes the definition of the abstract machine relatively simple, and it is enough to formally explain our unique *hybrid approach* to the design of the concurrency controller for rollback-free atomicity. The main feature of this approach is that it combines static (i.e. compile time) type checking with runtime versioning.

3.1.2 Typing for safe rollback-free atomicity

The first argument of the `atomic` $\overline{x}$ e construct specifies the input data required for the versioning algorithm to work correctly. Passing wrong data can jeopardize the isolation property. Thus, to make the language safe, we have proposed a type system that can statically verify if the data passed (dynamically) to the versioning algorithm will be correct. To our knowledge it was the first use of a type system for such an application. Our type system builds on Flanagan and Abadi's [31] type system for detection of race conditions. We have used their solution to ensure that all accesses to shared data are protected by locks. By refining the type system, this guarantee could be easily relaxed if needed. For instance, objects known to be immutable need not be visible to the concurrency controller when accessed, and so they could be left unprotected.

3.1.3 Isolation of operations with I/O effects

The main feature of our atomic tasks design is that it permits arbitrary sets of task critical operations to be isolated; in particular, operations with the I/O effects can also be part of the task's expression e. For this, they must be protected by *versioning locks* (or *verlocks*). Verlocks are similar to locks. However, the standard locking principle is extended with a runtime blocking for isolation preservation; blocking is guaranteed to be temporary up to the progress property of versioning algorithms.

The programmer can use a construct `sync` (implementing verlocks) to mark the task's operations that should be serialized. Other critical operations (not protected by verlocks) are not serialized. We have made this design choice since we wanted to be able to sometimes relax isolation. In some programs, strict task serialization can be too restrictive; e.g., some I/O actions should not be blocked. Alternatively, we could remove the `sync` construct from the front-end language (that is used by the programmers), and require that the compiler would place verlock-based synchronization automatically (based on the typing information). However, to be able to give a rigorous proof of isolation preservation, which included the proof of runtime versioning, we had to keep `sync` in the calculus. We assume that information on verlocks is provided in `atomic` explicitly, and leave type inference as an open problem.

The isolation guarantee in our language stems from three sources: (a) compile time enforcement that each shared data location (or an I/O operation) is protected by a verlock and that threads acquire the corresponding verlock before accessing the location (or performing the I/O operation), (b) compile

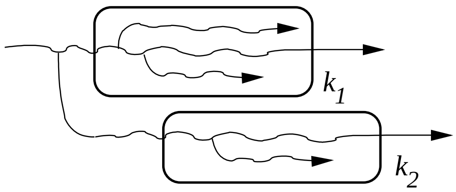


Figure 3.1: Concurrent, multithreaded atomic tasks

time enforcement that requires that all verlocks to be acquired during a task, are declared at the beginning of the task, and (c) a runtime locking strategy that assigns versions to threads that allow them to acquire verlocks so that isolation is preserved.

3.1.4 Task multithreading and nesting

An expression e of the task spawned with `atomic` $\overline{x}$ e, can be a concurrent program, i.e. atomic tasks can be multithreaded. For example, Figure 3.1 illustrates two concurrent, multithreaded tasks k_1 and k_2. Execution of each task is atomic with respect to other tasks that run on the same machine. However, the execution of concurrent threads inside each task can be arbitrarily interleaved. Threads in our language are lightweight processes. Threads of a single task can engage in two-way communication using shared data structures or I/O channels. Any synchronization required by this communication can be implemented using verlocks (or monitors, etc.).

When an atomic task k_1 is spawned with the `atomic` construct that is nested in some other atomic task (say k_2), then as long as there are no conflicts between critical operations of the two tasks, they can run in parallel. For this, every new atomic task is executed by a fresh thread. If some conflicts occur, then k_2 will be blocked till k_1 would release critical resources. Thus, in the (ideal) serial execution of the tasks, k_2 would commence after k_1 has completed (i.e. k_1 cannot see any effects of k_2). It follows from the definition in Section 2.1.2, that the isolation property of atomic tasks is satisfied. Note that this semantics is different from *nested transactions* [108] in databases. A nested transaction occurs when a new transaction is started inside an existing transaction. However, any changes made by the nested transaction are not seen by the parent transaction until the nested transaction is committed.

3.2 Example Program

Below is an example program, expressed using an object language (defined in Chapter 6) with `let`-binders from ML [75]. The program implements two concurrent tasks: k_1 and k_2. The first task (k_1) is performing a bank transfer: it withdraws an `amount` value from an account `accA`, and then it deposits the same value to an account `accB`. The second task (k_2) is computing the current balance `balance`, which must be equal the total amount of assets deposited by a client on accounts `accA` and `accB`. After the balance has been computed, the task outputs it on a channel `chan`; the output operation has an irrevocable effect of a message being sent out to the network.

```
let accA = new AccountA() in
let accB = new AccountB() in

newlock a : effect AccountA in
newlock b : effect AccountB in

atomic {a,b}    (* task k1 *)
(
  sync a accA.withdraw(amount);
  sync b accB.deposit(amount)
);

atomic {a,b}    (* task k2 *)
(
  let s = sync a accA.get() in
  let balance = s + sync b accB.get() in
  chan.output(balance)  (* irrevocable I/O effect *)
)
```

Note that the balance value computed by task k_2 would be diminished by `amount`, and so incorrect, if it would be computed just *after* the money had been withdrawn by task k_1 from one account but *before* they are deposited to another account. Fortunately, the isolation property of concurrent tasks guarantees that such erroneous interference of concurrent operations is forbidden; it ensures that any concurrent execution of atomic tasks is equivalent to an execution in which the tasks would be serialized. Thus, atomic tasks help in our program to maintain a consistent view of bank assets. Moreover, since the implementation of atomic tasks does not depend on rollback-recovery, `balance` is output to channel `chan` exactly once.

Execution of `newlock` $x : m$ `in` e in the above program creates a new versioning lock (verlock) x of type m that can be used in program e; the verlock type identifies data structures and I/O channels protected by the verlock. A fresh verlock is usually created for every communication channel and every data structure that can be shared by tasks. For instance, in our example program, verlocks have been created for the `a` and `b` objects of class types `AccountA` and `AccountB`.

Execution of `atomic` $\overline{e}$ e creates a new task for the evaluation of expression e. After the creation, e commences execution, in parallel with the rest of the body of the spawning program (each task is executed by a new thread). The $\overline{e}$ expression should give verlocks that can be used by a task to mark critical operations on data structures or I/O channels shared with other tasks. Tasks can spawn internal threads, which are executed within the scope of the task.

The `sync` construct can be used to mark critical operations of a task; the runtime system will guarantee that all operations marked in this way (and executed by any thread of the task) will be executed in isolation with respect to other concurrent tasks. Execution of `sync` e e' is similar to Java's `synchronized` statement [37], i.e. the expression e is evaluated first, and should yield a lock, which is then acquired when possible; the expression e' is then evaluated, giving a value v; and finally the lock is released and the value v is returned as the result of the whole expression.

What is a locking strategy in `sync` x e? Or, when exactly the verlock x is acquired when executing this expression? Verlock x of type m is *acquired* when two conditions are satisfied: (1) the effect m caused by e does not conflict (with respect to isolation) with any effects of other concurrent tasks, and (2) lock x is free (i.e. the standard locking principle applies). The second condition is required just to avoid races inside a task; it can be dropped if tasks are single-threaded.

The type system verifies if verlocks that may be acquired by a task spawned with `atomic` $\overline{e}$ e are known before the task commences, i.e. it checks if all such verlocks have been declared in $\overline{e}$. It thus eliminates errors due to omission of such declarations. For instance, the above program does not typecheck if the arguments `a` or `b` of `atomic` would be removed. Static verification (at compile time) ensures a safe execution of the concurrency controller at runtime.

Below we define our language formally, which has several advantages. Explaining the meaning of a programing language constructs using a natural language is very ambiguous. Therefore, we will present an operational semantics that is mathematically precise. The semantics has been split into a dynamic semantics of the host language constructs, and of the concurrency controller; we have used the semantics to formally prove correct the VA algorithm. We

$$
\begin{array}{llll}
\text{Variables} & x, y & \in \mathit{Var} & \\
\text{Type Var-s} & m, o & \in \mathit{TypVar} & \\
\text{Allocations} & a, b & \in 2^{\mathit{TypVar}} & \\
\text{Permissions} & p & \in 2^{\mathit{TypVar}} & \\
\text{Types} & s, t & & ::= \texttt{Unit} \mid t \to^{a,p} t \mid \texttt{Ref}_m\, t \mid m \\
\text{Values} & v, w & \in \mathit{Val} & ::= () \mid \lambda^{a,p} x : t.\, e \\
\text{Expressions} & e & \in \mathit{Exp} & ::= x \mid v \mid e\ e \mid \texttt{ref}_m\, e \mid\ !e \\
 & & \mid e := e \mid \texttt{newlock}\ x{:}m\ \texttt{in}\ e \mid \texttt{sync}\ e\ e & \\
 & & \mid \texttt{fork}\ e \mid \texttt{atomic}\ \overline{e}\ e &
\end{array}
$$

We work up to alpha-conversion of expressions throughout, with x binding in e in expressions $\lambda x : t.\, e$.

Figure 3.2: The calculus of atomic tasks: Syntax

have also shown several results (theorems) about our unique type-directed approach to concurrency control of rollback-free atomic tasks. The main result is that well-typed programs satisfy the isolation property. In the Appendix, we give a rigorous proof of isolation preservation and progress for our language (up to deadlocks between threads of the same task); the proof makes data and I/O accesses explicit and deals with multiple threads within an atomic block. This is one of the first such proofs for atomic tasks.

3.3 The Calculus of Atomic Tasks

3.3.1 Syntax

In this section we define the *calculus of atomic tasks* (or the *iso-calculus*, in short) as the call-by-value λ-calculus, extended with reference cells, atomic tasks, and versioning locks. The abstract syntax is in Figure 3.2. The main syntactic categories are values and expressions. We write $\overline{x}$ as shorthand for a possibly empty sequence of variables $x_1, ..., x_n$ (and similarly for $\overline{t}$, $\overline{e}$, etc.).

Types

Types include the base type `Unit` of unit expressions, which abstracts away from concrete ground types for basic constants (integers, Booleans, etc.), the type $t \to^{a,p} t$ of functions, the type $\texttt{Ref}_m\, t$ of reference cells containing a value of type t, and finally a singleton lock type m. A *singleton lock type* is the type of a single lock. The types of references and functions are decorated by correspondingly, m and a, p, where m is a singleton lock type of a verlock used to protect the reference cell against simultaneous accesses by concurrent threads, and a and p describe an *allocation* and *permission*. Allocations and permissions are sets of singleton lock types, representing respectively, the set of all verlocks that may be *demanded* during evaluation of a function, and the set of verlocks that must be *held* before a function call.

Values and expressions

A value is either an empty value () of type `Unit`, or function abstraction $\lambda^{a,p}x : t.\, e$ (decorated with allocation a and permission p). Values are first-class programming objects, they can be passed as arguments to functions and returned as results and stored in reference cells. Basic expressions e are mostly standard and include variables, values, function applications, reference creation $\texttt{ref}_m\, e$ (decorated with a singleton lock type m), and the usual imperative operations on references, i.e. dereference $!e$ and assignment $e := e$. We also assume existence of `let`-binders, and use syntactic sugar $e_1; e_2$ (sequential execution) for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ (for some x, where x is fresh).

Threads and atomic tasks

The language allows multithreaded programs by including the expression `fork` e, which spawns a new thread for the evaluation of expression e. This evaluation is performed only for its effect; the result of e is never used. Execution of $\texttt{atomic}\ \overline{e}\ e$ creates a new atomic task for the evaluation of expression e; a new thread is implicitly created for the task's execution. Tasks can use `fork` to spawn their own threads. The declaration $\overline{e}$ should give verlocks that can be used by a task to control access to shared data. All program threads will be interleaved while providing the illusion that tasks are executed in isolation.

Verlocks

The execution of $\texttt{newlock}\ x : m\ \texttt{in}\ e$ creates a new unique name x of a versioning lock (or verlock). It also introduces the type variable m which denotes the singleton lock type of the newly created verlock. Both x and m may be referred to in the expression e, i.e. x and m are bound in e. The expression $\texttt{sync}\ e\ e'$ is similar to Java's `synchronized` statement [37], i.e. the expression e is evaluated first, and should yield a verlock, which is then acquired when possible; the expression e' is then evaluated; and finally the verlock is released. Verlocks combine a simple lock (mutex) for protection against simultaneous data accesses by concurrent threads, with an algorithm that schedules lock

State Space

$$
\begin{array}{rcllll}
S & \in & \mathit{State} & = & \mathit{LockStore} \times \mathit{RefStore} \times \mathit{ThreadSeq} \\
\pi & \in & \mathit{LockStore} & = & \mathit{LockLoc} \to \{0,1\} \\
\sigma & \in & \mathit{RefStore} & = & \mathit{RefLoc} \to \mathit{Val} \\
l & \in & \mathit{LockLoc} & \subset & \mathit{Var} \\
r & \in & \mathit{RefLoc} & \subset & \mathit{Var} \\
pv & \in & \mathit{VerMap} & \subset & \mathit{LockLoc} \to \mathbf{Nat} \\
gv & \in & \mathit{VerMap} & \subset & \mathit{LockLoc} \to \mathbf{Nat} \\
lv & \in & \mathit{VerMap} & \subset & \mathit{LockLoc} \to \mathbf{Nat} \\
T & \in & \mathit{ThreadSeq} & ::= & f \mid T, T \\
f & \in & \mathit{Exp}_{ext} & ::= & x \mid v \mid f\ e \mid v\ f \\
 & & & & \mid \mathtt{ref}_m\, f \mid \mathtt{!}f \mid f := e \mid r := f \\
 & & & & \mid \mathtt{newlock}\ x\!:\!m\ \mathtt{in}\ e \mid \mathtt{sync}\ f\ e \mid \mathtt{insync}\ l\, f \\
 & & & & \mid \mathtt{fork}\ e \mid \mathtt{atomic}\ \overline{f}\overline{e}\ e \mid \mathtt{atomic}\ \overline{l}\overline{f}\ e \mid \mathtt{task}\ pv\ T
\end{array}
$$

Evaluation Contexts

$$
\begin{array}{rcl}
\mathcal{E} & = & [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \\
 & & \mid \mathtt{ref}_m\, \mathcal{E} \mid \mathtt{!}\mathcal{E} \mid \mathcal{E} := e \mid r := \mathcal{E} \\
 & & \mid \mathtt{sync}\ \mathcal{E}\ e \mid \mathtt{insync}\ l\, \mathcal{E} \\
 & & \mid \mathtt{atomic}\ \overline{l}\mathcal{E}\overline{e}\ e \mid \mathtt{task}\ pv\ \mathcal{E} \mid \mathcal{E}, T \mid T, \mathcal{E}
\end{array}
$$

Figure 3.3: The *iso*-calculus: Reduction semantics – Part I

acquisitions by (threads of) atomic tasks based on access versions; the details of the algorithm will be given in Section 3.3.3.

3.3.2 Operational semantics

We specify the operational semantics using the rules defined in Figure 3.3, 3.4, and 3.5. A state S consists of three elements: a lock store π and a reference store σ, which are sometimes referred to collectively as a store π, σ, and a collection of expressions T, which are organized as a sequence $T_0, ..., T_n$. Each expression T_i in the sequence represents a *thread*.

The *lock store* π is a finite map (or dictionary) from lock locations to their states; a lock location has two states, unlocked (0) and locked (1), and is initially unlocked. The *reference store* σ is a finite map from reference locations to values stored in the references. Lock locations l and reference locations r

Structural Congruence

$$T, T' \equiv T', T$$

$$T, () \equiv T$$

$$\frac{\langle \pi, \sigma \mid T \rangle \longrightarrow \langle \pi', \sigma' \mid T' \rangle}{\langle \pi, \sigma \mid \mathcal{E}[\,T\,] \rangle \longrightarrow \langle \pi', \sigma' \mid \mathcal{E}[\,T'\,] \rangle}$$

$$\frac{T \longrightarrow T'}{\langle \pi, \sigma \mid T \rangle \longrightarrow \langle \pi, \sigma \mid T' \rangle}$$

Transition Rules

$$\begin{array}{rcl} \mathit{eval} & \subseteq & \mathit{Exp} \times \mathit{Val} \\ \mathit{eval}(e, v_0) & \Leftrightarrow & \langle \emptyset, \emptyset \mid e \rangle \longrightarrow^* \langle \pi, \sigma \mid v_0, (), \cdots, () \rangle \end{array}$$

$$\lambda x.\, e\ v \longrightarrow e\{v/x\} \qquad \text{(R-App)}$$

$$\frac{r \notin \mathit{dom}(\sigma)}{\langle \pi, \sigma \mid \mathtt{ref}_m\, v \rangle \longrightarrow \langle \pi, (\sigma, r \mapsto v) \mid r \rangle} \qquad \text{(R-Ref)}$$

$$\langle \pi, \sigma \mid !r \rangle \longrightarrow \langle \pi, \sigma \mid v \rangle \quad \text{if} \ \ \sigma(r) = v \qquad \text{(R-Deref)}$$

$$\langle \pi, \sigma \mid r := v \rangle \longrightarrow \langle \pi, \sigma[r \mapsto v] \mid () \rangle \qquad \text{(R-Assign)}$$

$$\mathcal{E}[\,\mathtt{fork}\ e\,] \longrightarrow \mathcal{E}[\,()\,], e \qquad \text{(R-Fork)}$$

$$v_i, v'_j \longrightarrow v_i \quad \text{if} \ \ i < j \qquad \text{(R-Thread)}$$

$$\frac{\pi(l) = 1}{\langle \pi, \sigma \mid \mathtt{insync}\ l\, v \rangle \longrightarrow \langle \pi[l \mapsto 0], \sigma \mid v \rangle} \qquad \text{(R-InSync)}$$

Figure 3.4: The *iso*-calculus: Reduction semantics – Part II

are simply special kinds of variables that can be bound only by the respective stores (see Figure 3.3).

The expressions f are written in the calculus presented in Section 3.3.1, extended with a new construct `task` $pv\ T$, where pv is a map of *private version counters*, initialized to 0. The construct is not part of the language to be used by programmers; it will be used later to explain semantics.

We define a small-step evaluation relation $\langle \pi, \sigma \mid e \rangle \longrightarrow \langle \pi', \sigma' \mid e' \rangle$, read "expression e reduces to expression e' in one step, with stores π, σ being transformed to π', σ'". We also use $\longrightarrow^*$ for a sequence of small-step reductions.

By *concurrent evaluation*, or *concurrent run*, we mean a sequence of small-step reductions in which the reduction steps can be taken by different threads with possible interleaving.

Reductions are defined using evaluation context $\mathcal{E}$ for expressions e and f. The evaluation context ensures that the left-outermost reduction is the only applicable reduction for each individual thread in the entire program. Context application is denoted by [], as in $\mathcal{E}[\,e\,]$. Structural congruence rules allow us to simplify reduction rules by removing the context whenever possible.

The evaluation of a program e starts in an initial state with empty stores $(\emptyset, \emptyset)$ and with a single thread that evaluates the program's expression e. Evaluation then takes place according to the transition rules in Figure 3.4 and 3.5. The evaluation terminates once all threads have been reduced to values, in which case the value v_0 of the initial, first thread T_0 is returned as the program's result (typing will ensure that other values are empty values). Subscripts in values reduced from threads denote the sequence number of the thread, i.e. v_i is reduced from i's thread, denoted T_i $(i = 0, 1..)$.

The execution of threads can be arbitrarily interleaved. Since different interleavings may produce different results, the evaluator $eval(e, v_0)$ is therefore a relation, not a partial function.

Below we describe reduction rules in Figure 3.4. These rules are common for all versioning concurrency control algorithms, while rules in Figure 3.5, that will be described later, define our example versioning algorithm. The first four evaluation rules are the standard rules of a call-by-value λ-calculus, extended with references.

We write $\{v/x\}e$ to denote the capture-free substitution of v for x in the expression e. The notation $(\sigma, r \mapsto v)$ means "the store that maps r to v and maps all other locations to the same thing as σ".

Rules (R-Ref), (R-Deref), and (R-Assign) correspondingly, create a new reference cell with a store location r initially containing v, read the current store value, and assign a new value to the store located by r. For instance, let us look at the rule (R-Assign). We use the notation $\sigma[r \mapsto v]$ to denote update of map σ at r to v. Note that the term resulting from this evaluation step is just $()$; the interesting result is the updated store.

An expression f *accesses* a reference location r if there exists some evaluation context $\mathcal{E}$ such that $f = \mathcal{E}[\,!r\,]$ or $f = \mathcal{E}[\,r := v\,]$. (Note that both assign and dereference operations are non-commutative.)

Evaluation of expression `fork` e in (R-Fork) creates a new thread which evaluates e. The result of evaluating expression e is discarded by rule (R-Thread).

A program *completes*, or *terminates*, if all its threads reduce to a value. By (R-Thread), values of more recent threads are ignored, so that eventually only the value of the first thread T_0 will be returned by a program.

3.3.3 The Versioning Algorithm (VA)

The abstract machine of our language implements a runtime locking strategy for atomic tasks. The strategy essentially assigns "tickets" to task threads that allow them to acquire verlocks. The tickets are monotonically increasing version counters, one per verlock. On task entry, the task's thread obtains incremented ticket values (or "versions") for all the verlocks that it wants to acquire during the task. The thread can then acquire a verlock only when the corresponding verlock service count has reached its ticket count. Since tickets for all verlocks are obtained atomically, this guarantees that tasks with conflicts (shared verlocks) will acquire these shared verlocks in global order of task starts. By using the locking strategy, the concurrent execution of atomic tasks can satisfy the isolation property.

Below we describe the *Versioning Algorithm (VA)* that implements the above strategy. In order to keep the abstract machine simple, we have chosen one of the simplest versioning algorithms possible. It is very similar to the BVA algorithm, described in Chapter 2. Alternatively, we could adapt, e.g. the SVA or RVA algorithm, also described in that chapter. They permit more concurrency than the BVA but they are more complex.

Before we present the VA algorithm, we need to define some data structures that are required by the algorithm. Firstly, the program state is extended with a map gv of *global version counters* $gv(l)$ for each lock l in π (initialized to 0). A *version* is a natural number playing a role of access capability. Secondly, each lock l maintains a local version counter $lv(l)$, which is also initialized to 0; a map lv of *local version counters* is part of the state, too. We write gv_l and lv_l as shorthand for $gv(l)$ and $lv(l)$. For clarity we usually omit the counters in the rules when possible. The VA algorithm maintains an invariant that a local version of each lock is equal or less than a global version of the lock, and it is equal or greater than zero.

Transition Rules (of the Versioning Algorithm)

$$gv \in \mathit{VerMap} \subset \mathit{LockLoc} \rightarrow \mathbf{Nat}$$

$$lv \in \mathit{VerMap} \subset \mathit{LockLoc} \rightarrow \mathbf{Nat}$$

$$\mathit{eval}(e, v_0) \Leftrightarrow \langle \emptyset, \emptyset, \emptyset, \emptyset \mid e \rangle \longrightarrow^* \langle \pi, \sigma, gv, lv \mid v_0, (), \cdots, () \rangle$$

$$\frac{\begin{array}{c}\pi(l) \in \{0, 1\} \\ gv(l) \geq lv(l) \geq 0 \quad \text{for all} \quad l \in \mathit{dom}(\pi)\end{array}}{\langle \pi, \sigma, gv, lv \mid e \rangle \longrightarrow \langle \pi', \sigma', gv', lv' \mid e' \rangle} \qquad \text{(Invar)}$$

$$\mathit{VA0}: \quad \frac{\begin{array}{c}l \notin \mathit{dom}(\pi) \\ gv' = (gv, l \mapsto 0) \quad lv' = (lv, l \mapsto 0)\end{array}}{\begin{array}{c}\langle \pi, \sigma, gv, lv \mid \texttt{newlock}\ x\!:\!m\ \texttt{in}\ e \rangle \longrightarrow \\ \langle (\pi, l \mapsto 0), \sigma, gv', lv' \mid e\{l/x\}\{o_l/m\} \rangle\end{array}} \qquad \text{(R-Lock)}$$

$$\mathit{VA1}: \quad \frac{\begin{array}{c}\bar{l} = l_1, ..., l_n \\ gv' = gv[l_i \mapsto gv(l_i) + 1] \quad i = 1..n \\ pv = (l_1 \mapsto gv'(l_1), ..., l_n \mapsto gv'(l_n))\end{array}}{\begin{array}{c}\langle \pi, \sigma, gv, lv \mid \mathcal{E}[\, \texttt{atomic}\ \bar{l}\ e \,] \rangle \longrightarrow \\ \langle \pi, \sigma, gv', lv \mid \mathcal{E}[\, () \,], \texttt{task}\ pv\ e \rangle\end{array}} \qquad \text{(R-Isol)}$$

$$\texttt{task}\ pv\ \mathcal{E}[\, \texttt{fork}\ e \,] \longrightarrow \texttt{task}\ pv\ (\mathcal{E}[\, () \,], e) \qquad \text{(R-Fork')}$$

$$\mathit{VA2}: \quad \frac{\pi(l) = 0 \quad pv(l) - 1 = lv(l)}{\begin{array}{c}\langle \pi, \sigma, gv, lv \mid \texttt{task}\ pv\ \mathcal{E}[\, \texttt{sync}\ l\ e \,] \rangle \longrightarrow \\ \langle \pi[l \mapsto 1], \sigma, gv, lv \mid \texttt{task}\ pv\ \mathcal{E}[\, \texttt{insync}\ l\, e \,] \rangle\end{array}} \qquad \text{(R-Sync)}$$

$$\mathit{VA3}: \quad \frac{\begin{array}{c}pv(l) - 1 = lv(l) \\ lv' = lv[l \mapsto pv(l)] \quad \text{for all} \quad l \in \mathit{dom}(pv)\end{array}}{\pi, \sigma, \langle gv, lv \mid \texttt{task}\ pv\ v \rangle \longrightarrow \pi, \sigma, \langle gv, lv' \mid () \rangle} \qquad \text{(R-Task)}$$

Figure 3.5: The *iso*-calculus: Reduction semantics – Part III

Definition 4 (Versioning Algorithm (VA))**.** The algorithm is given by the following set of rules or steps:

VA0: Upon lock creation, initialize global and local counters of the new lock to zero.

VA1: At the moment of spawning a new task k using `atomic` $\bar{l}$ e, for each lock l_i where $i = 1, .., |\bar{l}|$, that may be requested by this task, increase counter gv_{l_i} by one. Create a fresh (read-only) map pv_k that contains bindings from the locks l_i to their upgraded versions gv_{l_i}.

VA2: A task k can acquire a lock l only when, the lock is free and the task holds a (private) version of this lock that—when downgraded by one—matches the current (local) version maintained by the lock, i.e.

$$pv_k(l) - 1 = lv_l \quad . \tag{3.1}$$

VA3: After a task k has completed its execution, i.e. all threads of the task have terminated, for each lock l_i, where $i = 1, .., |\bar{l}|$, wait until condition (3.1) is true, then upgrade a local version of each lock l_i, so that $lv_{l_i} = pv_k(l_i)$.

We require steps *VA1* and *VA2* to be critical sections (atomic).

The steps of the VA algorithm essentially define task creation and destruction, and verlock acquisition and release. Thus, we can define the algorithm precisely using the operational semantics of our language. The reduction rules corresponding to the above steps *VA0–3* are given in Figure 3.5; these are, respectively, (R-Lock), (R-Isol), (R-Sync), and (R-Task). Also, the invariant maintained by the algorithm is defined using a separate rule (Invar). Below we explain some of these reduction rules in more detail.

Task creation and destruction

Evaluation of a term `atomic` $\bar{l}$ e creates a new thread for evaluation of expression `task` pv e; see (R-Isol). The term `task` pv e is an *atomic task* evaluating e, where pv is a *private versions* map of (ver)locks $\bar{l}$ declared in `atomic`. The map pv associates lock locations with globally unique versions, maintained by global version counters gv. The map pv is created for a given set of (ver)locks dynamically in one atomic step, and remains constant for the task's lifetime. Program evaluation maintains an invariant that a private version of each lock in a private versions map of every task is globally unique.

Tasks can spawn their own threads using `fork`; see (R-Fork'). Tasks and threads are used only for their side-effects, which are in our case modifications to the store; an output to the store can be regarded as an operation with either a memory or I/O effect.

An atomic task `task` $pv\ e$ has *completed* (or *terminated*) its execution if expression e yields a value; see (R-Task). Then the task upgrades local counters of its verlocks and reduces to an empty value. (The two variants of the VA algorithm that we have described in Chapter 2, permit more concurrency by making these upgrades *during* task execution, and so releasing shared resources earlier.) To ensure that the order of upgrades by all tasks is correct, the task completion is guarded by the condition that $pv(l) - 1$ must be equal $lv(l)$ for all l in $dom(pv)$.

A state S is *task-free* if it does not have a context $\mathcal{E}[\,\texttt{task}\ pv\ T\,]$. Any task-free state is called a *result state*. The result states subsume data stored in all reference cells.

Serialized and isolated task evaluation

Two tasks are executed *serially* if one task commences after another one has completed. By *serial evaluation*, or *serial run*, we mean evaluation, in which all tasks are executed serially. (Note that a serial run is also concurrent since serialized tasks may be themselves multithreaded.)

Isolation has been proposed as the correctness condition of concurrency control algorithms for atomic transactions in database systems [9]. It means, intuitively, that if the effects of one transaction are visible to some other transaction executing concurrently, then the opposite is not true, where an *effect* is usually defined as any change to the content of shared data structures (which are modelled in our language using reference cells); from the perspective of a transaction, it appears that atomic transactions execute sequentially rather than in parallel.

In our language, we have extended the above definition of an effect, and assume that both assignment and dereference has an effect, respectively an *output* and an *input* effect. (In database systems, usually only a write operation, i.e. assignment, has an effect.) Therefore, our definition of isolation will be more conservative than in [9]. It can be captured precisely using the notion of task noninterference, defined as follows.

Tasks in a concurrent run *do not interfere* (or satisfy the *noninterference* property) if there exists some ideal serial run R^{s} of all the tasks such that given any reference, the order of accessing the reference by tasks in the concurrent run is the same as in R^{s}. By the *isolated evaluation* of an expression (containing some tasks) we mean any evaluation of this expression that satisfies the isolation property, defined as follows.

Definition 5 (Isolation Property)**.** Evaluation of an expression e satisfies the *isolation property* if all tasks of e do not interfere. A given program satisfies the isolation property if all its terminating evaluations satisfy this property.

Verlock acquisition and release

The expression `newlock` $x : m$ `in` e (see rule (R-Lock)) dynamically creates a new verlock's lock location l (with the initial state 0), extending global versions gv, local versions lv, and a lock store π accordingly, and replaces the occurrences of x in e with l. It also replaces occurrences of m in e with a type variable o_l that denotes the corresponding singleton lock type.

A lock store π that binds a verlock's lock location l also implicitly binds the corresponding type variable o_l with kind `Lock`; the only value of o_l is l. Below we sometimes confuse a verlock and the verlock's lock location, where it is clear from the context what we mean.

A lock location l is *free* if $\pi(l) = 0$, otherwise it is not free.

The semantics of `sync` $e\ e'$ executed by a task is defined by rule (R-Sync). The expression e is evaluated first, and should yield a verlock l, which is then acquired ($\pi(l) = 1$) if free *and* if the task holds a version number pv for l that matches a local version maintained by l (i.e. $pv(l) - 1 = lv(l)$). The expression e' is then evaluated as part of an expression `insync` $l\, e'$. The verlock is released ($\pi(l) = 0$) by (R-InSync) when the expression e' reduces to a value v (then `insync` $l\, v$ is replaced by v).

The second premise of rule (R-Sync) ($pv(l) - 1 = lv(l)$) guarantees that a task can acquire a verlock only at a time when it is *safe*, i.e. when accessing data protected by the verlock does not invalidate isolation. Otherwise, the verlock's lock is not taken even if it is free, resulting in the task's thread being blocked (any other threads are not blocked).

Each verlock's lock requested by a task will be eventually acquired (evaluation progress) if only tasks are themselves deadlock-free and terminate. We discuss the deadlock issue in Section 3.4.3, after explaining typing.

Correctness assumptions

The VA algorithm guarantees noninterference, provided the following two conditions hold. Firstly, programs do not have race conditions, i.e. no data can be accessed without first acquiring a verlock. Secondly, all verlocks that *may* (not necessarily have to) be used by a task are known at a time when the task is spawned, so that the (R-Isol) rule can create the private version for each such verlock type, stored in the task's map *pv*. To maximize parallelism, we require only such verlocks to be declared. In Section 3.3.4 we define typing rules intended for checking statically if these two conditions hold in programs expressed using our language. Then, we show in Section 3.4 that our type system is sound, making the language *safe by construction*.

3.3.4 Typing

The type system is formulated in Figure 3.6 as a deductive proof system, defined using conclusions (or judgments) and the static inference rules for reasoning about the judgments. The typing judgment for expressions has the form $\Gamma; a; p \vdash e : t$, read "expression e has type t in environment Γ with allocation a and permission p", where an environment Γ is a finite mapping from free variables to types. An expression e is a *well-typed program* if it is closed and it has a type t in the empty type environment, written $\vdash e : t$.

Our intend is that, if the judgment $E; a; p \vdash e : t$ holds, then any terminating execution of expression e is race-free, satisfies the isolation property, and yields values of type t, provided:

(i) the current thread holds at least the verlocks described by permission p (Condition 1),

(ii) if expression e is part of a task, then the task has declared all verlocks described by allocation a (Condition 2), and

(iii) the free variables of e are given bindings consistent with Γ.

We will show in Section 3.4 that the type system is sound. Based on this result, we state dynamic correctness of our example concurrency control algorithm, which together with type soundness gives the expected result of isolation preservation.

Our type system is an extension of Flanagan and Abadi's type system for detecting race conditions [31]. It provides rules for proving that the above two conditions are always true for well-typed programs. Condition 1 is verified using an approach described in [31]. The set of typing rules in Figure 3.6 has been obtained by extending this approach with allocations needed to verify Condition 2, and adding a new rule for typing the `atomic` construct. Most of the typing rules are fairly straightforward. For simplicity, we present a first-order type system and omit subtyping of allocations. The subtyping rules would be similar to the subtyping rules with permissions in [31], where also extensions with polymorphism and existential types have been described.

Judgments

$\Gamma \vdash \diamond$	Γ is a well-formed typing environment
$\Gamma \vdash t$	t is a well-formed type in Γ
$\Gamma \vdash a, p$	a, p is a well-formed resource allocation and permission in Γ
$\Gamma; a; p \vdash e : t$	e is a well-typed expression of type t in Γ with allocation a and permission p

Typing Rules

$$\frac{}{\emptyset \vdash \diamond} \quad \text{(Env-}\emptyset\text{)}$$

$$\frac{\Gamma \vdash t \quad x \notin dom(\Gamma)}{\Gamma, x : t \vdash \diamond} \quad \text{(Env-}x\text{)}$$

$$\frac{\Gamma \vdash \diamond \quad m \notin dom(\Gamma)}{\Gamma, m :: \texttt{Lock} \vdash \diamond} \quad \text{(Env-}m\text{)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{Unit}} \quad \text{(Type-Unit)}$$

$$\frac{\Gamma \vdash t \quad \Gamma \vdash t' \qquad \Gamma \vdash a, p}{\Gamma \vdash t \rightarrow^{a,p} t'} \quad \text{(Type-Fun)}$$

$$\frac{\Gamma \vdash t \quad \Gamma \vdash m}{\Gamma \vdash \texttt{Ref}_m\, t} \quad \text{(Type-Ref)}$$

$$\frac{m :: \texttt{Lock} \in \Gamma}{\Gamma \vdash m} \quad \text{(Type-Lock)}$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash m \qquad \text{for all } \; m \in a \cup p}{\Gamma \vdash a, p} \quad \text{(Alloc)}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma; a; p \vdash () : \texttt{Unit}} \quad \text{(T-Unit)}$$

$$\frac{x : t \in \Gamma}{\Gamma; a; p \vdash x : t} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x : s; a; p \vdash e : t}{\Gamma; a'; p' \vdash \lambda^{a,p} x : s.\, e : s \rightarrow^{a,p} t} \quad \text{(T-Fun)}$$

$$\frac{\Gamma; a; p \vdash e : s \rightarrow^{a',p'} t \qquad \Gamma; a; p \vdash e' : s \quad a' \subseteq a \quad p' \subseteq p}{\Gamma; a; p \vdash e\ e' : t} \quad \text{(T-App)}$$

$$\frac{\Gamma \vdash m \quad \Gamma; a; p \vdash e : t}{\Gamma; a; p \vdash \texttt{ref}_m\, e : \texttt{Ref}_m\, t} \quad \text{(T-Ref)}$$

$$\frac{\Gamma; a; p \vdash e : \texttt{Ref}_m\, t \quad m \in p}{\Gamma; a; p \vdash !e : t} \quad \text{(T-Deref)}$$

$$\frac{\Gamma; a; p \vdash e : \texttt{Ref}_m\, t \qquad \Gamma; a; p \vdash e' : t \quad m \in p}{\Gamma; a; p \vdash e := e' : \texttt{Unit}} \quad \text{(T-Assign)}$$

$$\frac{\Gamma, m :: \texttt{Lock}, x : m; a; p \vdash e : t \qquad \Gamma \vdash a, p \quad \Gamma \vdash t}{\Gamma; a; p \vdash \texttt{newlock}\ x{:}m\ \texttt{in}\ e : t} \quad \text{(T-Lock)}$$

$$\frac{\Gamma; a; p \vdash e : m \quad m \in a \qquad \Gamma; a; p \cup \{m\} \vdash e' : t}{\Gamma; a; p \vdash \texttt{sync}\ e\ e' : t} \quad \text{(T-Sync)}$$

$$\frac{\Gamma; a; \emptyset \vdash e : \texttt{Unit}}{\Gamma; a; p \vdash \texttt{fork}\ e : \texttt{Unit}} \quad \text{(T-Fork)}$$

$$\frac{\Gamma; a; p \vdash e_i : m_i \;\; \text{for all} \;\; i = 1..|\overline{e}| \qquad \Gamma; \{m_1\} \cup ... \cup \{m_{|\overline{e}|}\}; \emptyset \vdash e_0 : t}{\Gamma; a; p \vdash \texttt{atomic}\ \overline{e}\ e_0 : \texttt{Unit}} \quad \text{(T-Isol)}$$

Figure 3.6: The first-order type system for the *iso*-calculus

To verify Conditions 1 and 2, a verlock l is represented at the type level with a singleton lock type m that contains l. The singleton type allows typing rules to assert that a thread holds verlock l by referring to that type rather than to the verlock l. During typechecking, each expression is evaluated in the context of allocations a and permissions p. Including a singleton lock type in the allocation a, respectively permission p, amounts to assuming that the corresponding verlock's version, respectively the corresponding verlock, are held during the evaluation of e.

For instance, consider typing dereference and assignment operations on references, as part of typechecking some expression e''. As in [31], the corresponding rules (T-Deref) and (T-Assign) check if a singleton lock type m decorating the reference type is among lock types mentioned in the current permission p. The permission p can be extended with m only while typechecking a synchronization expression `sync` $e\ e''$, where e has type m (see typing of e in (T-Sync)).

To verify if a task e_0 executing `sync` $e\ e'$ has declared verlock e of some type m, we introduce an allocation a and require that m is mentioned in a. Note that m can be added to allocation a only while typechecking the construct `atomic` that has spawned task e_0. The rule (T-Isol) creates the allocation a from singleton types of all verlocks declared by the task; the allocation is then used for typechecking the body of the task.

An allocation a and permission p decorate a function type and function definition, representing respectively, allocation a—the set of all verlocks that may be requested while evaluating the function and any thread spawned by it, and permission p—the set of verlocks that must be held before a function call. Note that allocations are preserved by thread spawning since we allow tasks to be multithreaded, while permissions are nulled since spawned threads do not inherit locks from their parent thread.

Rules (T-Fork) and (T-Isol) require the type of the whole expression to be `Unit`; this is correct since threads are evaluated only for their side-effects.

3.4 Type System

The fundamental property of the type system and abstract machine of our language is that evaluation of well-typed, terminating programs satisfies the isolation property. The first component of the proof of this property is a *type preservation* result, stating that typing is preserved during evaluation. The second one is a *progress* result, stating that evaluation of an expression never enters into a state for which there is no evaluation rule defined. To prove both

Judgments

$\vdash S : t$ $\quad$ S is a well-typed state of type t

Rules

$$\frac{\Sigma(l) = \{0, 1\} \quad \Sigma(o_l) = \texttt{Lock}}{\Sigma \mid \Gamma; a; p \vdash l : o_l} \quad \text{(T-LockLoc)}$$

$$\frac{\Gamma \vdash m \quad \Sigma(r) = t}{\Sigma \mid \Gamma; a; p \vdash r : \texttt{Ref}_m\, t} \quad \text{(T-RefLoc)}$$

$$\frac{\begin{array}{c} dom(\pi) = \{l_1, ..., l_j\} \quad dom(\sigma) = \{r_1, ..., r_k\} \\ \Sigma = l_1 : \{0, 1\}, ..., l_j : \{0, 1\}, r_1 : s_1, ..., r_k : s_k, \\ o_{l_1} :: \texttt{Lock}, ..., o_{l_j} :: \texttt{Lock} \\ |T| > 0 \quad \Sigma \mid \Gamma; a_i; p_i \vdash T_i : t_i \quad \text{for all} \;\; i < |T| \end{array}}{\vdash \langle \pi, \sigma \mid T \rangle : t_0} \quad \text{(T-State)}$$

$$\frac{\vdash S : t_0 \quad \vdash S' : t_0}{\vdash S + S' : t_0} \quad \text{(T-Choice)}$$

$$\frac{\begin{array}{c} \Sigma \mid \Gamma; a; p \vdash f_i : t_i \\ \Sigma \mid \Gamma; a'; p' \vdash f'_j : t_j \quad i < j \end{array}}{\Sigma \mid \Gamma; a; p \vdash f_i, f'_j : t_i} \quad \text{(T-Thread)}$$

$$\frac{\begin{array}{c} a = \{o_{l_1}, ..., o_{l_n}\} \quad \Sigma \mid \Gamma; a; p \vdash l_i : o_{l_i} \\ \Sigma \mid \Gamma; a; p \vdash pv(l_i) : \mathbf{Nat} \quad \text{for all} \;\; i = 1..n \\ \Sigma \mid \Gamma; a; p \vdash T : t \end{array}}{\Sigma \mid \Gamma; a; p \vdash \texttt{task}\; pv\; T : \texttt{Unit}} \quad \text{(T-Task)}$$

$$\frac{\begin{array}{c} \Sigma \mid \Gamma; a; p \vdash l : m \\ \Sigma \mid \Gamma; a; p \vdash f : t \quad m \in a \quad m \in p \end{array}}{\Sigma \mid \Gamma; a; p \vdash \texttt{insync}\; l\, f : t} \quad \text{(T-InSync)}$$

$$\mathbf{Nat} = 0, 1, 2, .. \;\; \text{(includes zero)}$$

Figure 3.7: Additional judgments and rules for typing states

results, we extended typing judgments from expressions *Exp* to expressions Exp_{ext}, and then to states as shown in Figure 3.7. The judgment $\vdash S : t$ says that "S is a well-typed state yielding values of type t". We assume a single, definite type for every location in the store π, σ. These types have been collected as a *store typing* Σ—a finite function mapping locations to types, and type variables to kinds.

Type preservation and progress yield that our type system is *sound*. It guarantees that if a program is well-typed then:

(i) each operation on references requires to first obtain a verlock, and

(ii) if obtaining a verlock is part of some task spawned using the `atomic` construct, then the task has a private version of this verlock (which is possible only if the name of it is the argument of the construct).

The first property is called *absence of race conditions* and is guaranteed by Abadi and Flanagan's type system for avoiding race conditions that we have extended. The second property is called *absence of non-declared verlocks* and is guaranteed by our extension of their type system. Based on the two properties of the type system, we have proven that evaluation of well-typed, terminating programs satisfies the isolation property; the proof is in the Appendix (see also the technical report [113]).

Below we state formally the absence of race conditions and the absence of non-declared verlocks properties. Finally, we give our main result of isolation preservation in Section 3.4.3.

3.4.1 Absence of races

After removing allocations a and the rule (T-Isol) for typing the construct `atomic` in Figure 3.6, and replacing the semantics of verlocks by simple locks, we obtain Flanagan and Abadi's first-order type system [31]. The fundamental property of this type system is that well-typed programs do not have race conditions. Below are three Lemmas and one Theorem as found in [31], extended with store typing Σ and allocations that appear in our language; this small extension does not change any original proofs. We quote these lemmas and the theorem since they will be useful in our proof of type soundness.

The semantics can be used to formalize the notion of a race condition, as follows. A state has a *race condition* if its thread sequence contains two expressions that access the same reference location. A program e has a race condition if its evaluation may yield a state with a race condition, i.e. if there exists a state S such that $\langle \emptyset, \emptyset \mid e \rangle \longrightarrow^* S$ and S has a race condition.

Independently of the type system, locks provide mutual exclusion, in that two threads can never be in a critical section on the same lock. An expression f is in a *critical section* on a lock location l if $f = \mathcal{E}[\,\texttt{insync}\ l\, f'\,]$ for some evaluation context $\mathcal{E}$ and expression f'. The judgment $\vdash_{cs} S$ says that at most one thread is in a critical section on each lock in S. According to Lemma 4, the property $\vdash_{cs} S$ is maintained during evaluation.

Lemma 4 (Mutual Exclusion [31])**.**
If $\vdash_{cs} S$ and $S \longrightarrow S'$, then $\vdash_{cs} S'$.

Lemma 5 says that a well-typed thread accesses a reference cell only when it holds the protecting lock.

Lemma 5 (Lock-Based Protection [31])**.**
Suppose that $\Sigma \mid \Gamma; a; p \vdash f : t$, and f accesses reference location r. Then $\Sigma \mid \Gamma; a; p \vdash r : \texttt{Ref}_m\, t'$ for some lock type m and type t'. Furthermore, there exists lock location l such that $\Sigma \mid \Gamma; a; p \vdash l : m$ and f is in a critical section on l.

The lemma below implies that states that are well-typed and well-formed with respect to critical sections do not have race conditions.

Lemma 6 (Race-Conditions-Free States [31])**.** Suppose $\vdash S : t$ and $\vdash_{cs} S$. Then S does not have a race condition.

Finally, we can conclude that well-typed programs do not have race conditions.

Theorem 1 (Absence of Race Conditions [31])**.**
If $\vdash e : t$ then e does not have a race condition.

3.4.2 Absence of non-declared verlocks

An expression f is *part of* a task $\texttt{task}\ pv\ T$ if $T = \mathcal{E}[\,f\,]$ for some evaluation context $\mathcal{E}$. A task $\texttt{task}\ pv\ T$ *has a version* of a lock l if $pv(l)$ is defined. An expression f *has a version* of a lock l if there exists some task which has a version of l, and f is part of this task. An expression f *requests* a lock location l if $f = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some evaluation context $\mathcal{E}$ and expression e. A task $\texttt{task}\ pv\ T$ is in a *critical section* on a lock location l, if some thread of T is in a critical section on the lock location l.

Now, for the complete language with `atomic` and `task`, the judgment $\vdash_{cs} S$ says in addition to mutual exclusion property stated in Section 3.4.1, that each task being in a critical section on some lock in state S has a version of this

Judgments

$\mathcal{M} \vdash_{cs} f$	f has exactly one critical section for each lock in $\mathcal{M}$
$\mathcal{M} \vdash_{cs} \texttt{task}\ pv\ T$	task T has a version $pv(l)$ for each lock l in $\mathcal{M}$
$\vdash_{cs} S$	S is well-formed with respect to critical sections and tasks
$\vdash_{tf} S$	S is well-formed and task-free

Rules for Critical Sections of [31]

$$\frac{f = x \mid v \mid \texttt{newlock}\ x\!:\!m\ \texttt{in}\ e \mid \texttt{fork}\ e}{\emptyset \vdash_{cs} f} \quad \text{(CS-Empty)}$$

$$\frac{\begin{array}{c}\mathcal{M} \vdash_{cs} f \\ f' = f\ e \mid v\ f \mid \texttt{ref}_m\, f \mid\ !f \\ \mid f := e \mid r := f \mid \texttt{sync}\ f\ e\end{array}}{\mathcal{M} \vdash_{cs} f'} \quad \text{(CS-Exp)}$$

$$\frac{\mathcal{M} \vdash_{cs} f}{\mathcal{M} \uplus \{l\} \vdash_{cs} \texttt{insync}\ l\, f} \quad \text{(CS-InSync)}$$

$$\frac{\begin{array}{c}\forall i < |T| \quad \mathcal{M}_i \vdash_{cs} T_i \\ \mathcal{M} = \mathcal{M}_0 \uplus \ldots \uplus \mathcal{M}_{|T|-1} \\ \forall l \in \mathcal{M} \quad \pi(l) = 1\end{array}}{\vdash_{cs} \langle \pi, \sigma \mid T \rangle} \quad \text{(CS-State)}$$

Figure 3.8: Judgments and rules for reasoning about critical sections

$$\frac{\begin{array}{c}\forall i = 1..|\overline{f}| \quad \mathcal{M}_i \vdash_{cs} f_i \\ \mathcal{M} = \mathcal{M}_1 \uplus \ldots \uplus \mathcal{M}_{|\overline{f}|} \\ f' = \texttt{atomic}\ \overline{f}\ e\end{array}}{\mathcal{M} \vdash_{cs} f'} \quad \text{(CS-Isol)}$$

$$\frac{\begin{array}{c}\forall i < |T| \quad \mathcal{M}_i \vdash_{cs} T_i \\ \mathcal{M} = \mathcal{M}_0 \uplus \ldots \uplus \mathcal{M}_{|T|-1} \\ \forall l \in \mathcal{M} \quad pv(l) \ \text{ is defined and } \ pv(l) > 0\end{array}}{\mathcal{M} \vdash_{cs} \texttt{task}\ pv\ T} \quad \text{(CS-Task)}$$

$$\frac{\begin{array}{c}\vdash_{cs} \langle \pi, \sigma \mid T \rangle \\ \forall i < |T| \quad T_i \neq \texttt{task}\ pv\ T'\end{array}}{\vdash_{tf} \langle \pi, \sigma \mid T \rangle} \quad \text{(TF-State)}$$

Figure 3.9: Additional rules for critical sections and task free states

lock (see Figures 3.8 and 3.9). According to Lemma 7, the property $\vdash_{cs} S$ is maintained during evaluation.

Lemma 7 (Version-Completeness Preservation)**.** If $\vdash_{cs} S$ and $S \longrightarrow S'$, then $\vdash_{cs} S'$.

Lemma 8 says that a well-typed thread obtains a verlock only when it holds a version of this verlock.

Lemma 8 (Version-Based Protection)**.**
Suppose that $\Sigma \mid \Gamma; a; p \vdash f : t$, and f requests a lock location l. Then $\Sigma \mid \Gamma; a; p \vdash l : m$ for some lock type m. Furthermore, there exists a task $\texttt{task}\ pv\ T$ which f is part of, such that $\Sigma \mid \Gamma; a; p \vdash \texttt{task}\ pv\ T : \texttt{Unit}$ and version $pv(l)$ is defined.

The above property implies that in our language all lock requests are part of some task. This feature has simplified the type system and reasoning about the isolation property. A full-size language could make a difference between accessing a lock as part of some task, or outside tasks.

We conclude that all verlocks used by each task in well-typed programs are known a priori.

Theorem 2 (Verlock-Usage Predictability)**.** All verlocks that may be requested by a task of a well-typed program are known before the task begins.

The above result implies that the VA algorithm will be able to create upon a task's creation, a private version of each verlock that may be used by the task.

3.4.3 Isolation preservation

We have defined the isolated evaluation for complete tasks (see Section 3.3.3). This is however not a problem since in practice we are interested only in result states of this evaluation. Below we therefore formulate an isolation preservation result for traces (i.e. sequences of evaluated states) that begin and finish in a task-free state. The judgment for such states has the form $\vdash_{tf} S$, read "state S is well-formed and task-free", which means that either no task has been spawned yet, or if there were any, then they have already completed.

Below we state that each trace of a well-typed program has the "isolation up to" property, provided that the corresponding evaluation finishes in a result state.

Lemma 9 (Isolation Property Up To)**.** Suppose $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash S : t$ and $\vdash_{tf} S$. If $S \longrightarrow^* S'$ and $\vdash_{tf} S'$, then the run $S \longrightarrow^* S'$ satisfies the isolation property up to S'.

Based on the above lemma, we can prove that well-typed, terminating programs satisfy the isolation property. A program *terminates* if all its runs terminate; a run *terminates* if it reduces to a value.

Theorem 3 (Isolation Property)**.** If $\vdash e : t$, then all terminating runs $e \longrightarrow^* v_0$, where v_0 is some value of type t, satisfy the isolation property.

Proof of Theorem 3 is based on *dynamic correctness* of the VA algorithm, formulated using the following theorem.

Theorem 4 (Noninterference)**.** If a program has properties (i) and (ii) (see Section 3.4, 2nd paragraph) then any evaluation of the program up to any result state, using the VA algorithm, satisfies the noninterference property.

Deadlocks

We stated our main result for terminating programs. Note however that if a program deadlocks or never terminates, all its runs reaching some result state have the "isolation up to" property (up to this state). Thus, the deadlock issue is orthogonal to the goals of our work, and can be solved using the existing approaches.

The only deadlocks possible in our language stem from either two threads of the same task trying to acquire two locks l_1 and l_2 in parallel but in a different order, or when a thread tries to acquire a lock again before releasing it. This means however that other tasks that want to acquire these locks will be also blocked. Deadlock can be avoided by imposing a strict partial order on verlocks within each task, and respecting this order when acquiring verlocks; our language and type system can be extended with this principle by embodying the solution described in [31].

3.4.4 Proving type soundness

Reduction of a program may either continue forever, or may reach a final state, where no further evaluation is possible. Such a final state represents either an answer or a type error. Since programs expressed in our language are not guaranteed to be deadlock-free, we also admit a deadlocked state to be an (acceptable) answer. Thus, proving type soundness means that well-typed programs yield only well-typed answers.

Our proof of type soundness rests upon the notion of type preservation (also known as subject reduction). The type preservation property states that reductions preserve the type of expressions. Below are excerpts from the proof; see the Appendix for the complete proof.

Type safety

The statement of the main type preservation lemma must take stores and store typings into account. For this we need to relate stores with assumptions about the types of the values in the stores. Below we define what it means for a store π, σ to be well typed. (For clarity, we omit permissions p from the context.)

Definition 6. A store π, σ is said to be *well typed* with respect to a store typing Σ and a typing context Γ, written $\Sigma \mid \Gamma; a \vdash \pi, \sigma$, if $dom(\pi, \sigma) = dom(\Sigma)$ and $\Sigma \mid \Gamma; a \vdash \mu(l) : \Sigma(l)$ for every store $\mu \in \{\pi, \sigma\}$ and every $l \in dom(\mu)$.

Intuitively, a store π, σ is consistent with a store typing Σ if every value in the store has the type predicted by the store typing.

Type preservation for our language states that the reductions defined in Figures 3.3, 3.4 and 3.5 preserve type:

Theorem 5 (Type Preservation)**.** If $\Sigma \mid \Gamma; a \vdash T : t$ and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ and $\langle \pi, \sigma \mid T \rangle \longrightarrow \langle (\pi, \sigma)' \mid T' \rangle$, then for some $\Sigma' \supseteq \Sigma$, $\Sigma' \mid \Gamma; a \vdash T' : t$ and $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$.

Evaluation progress

Subject reduction ensures that if we start with a typable expression, then we cannot reach an untypable expression through any sequence of reductions. This by itself, however, does not yield type soundness.

We also had to show that evaluation of a typable expression cannot get *stuck*, i.e. either the expression is a value or there is some reduction defined. However, we do allow reduction to be suspended indefinitely since our language is not deadlock-free. This is acceptable since we define and guarantee isolation, respectively isolation-up-to, only for programs that either terminate, or reach some result state (see Theorem 3 and Lemma 9).

We state progress only for closed expressions, i.e. with no free variables. For open terms, the progress theorem fails. This is however not a problem since complete programs—which are the expressions we actually care about evaluating—are always closed.

Independently of the type system and store typing, we should define which state we regard as well-formed. Intuitively, a state is well-formed if the content of the store is consistent with the expression executed by the thread sequence. In case of store π, if there is some evaluation context $\mathcal{E}[\,\texttt{insync}\ l\,e\,]$ in the thread sequence for any lock location l, then $\pi(l)$ should contain 1, marking that the lock has been acquired. As for the store σ, containing the content of each reference cell, we may only require that it is well typed.

Definition 7. Suppose π, σ is a well-typed store, and $\overline{f}$ is a well-typed sequence of expressions, where each expression is evaluated by a thread. Then, a state $\pi, \sigma \mid \overline{f}$ is *well-formed*, denoted $\vdash_{wf} \pi, \sigma \mid \overline{f}$, if for each expression f_i ($i < |\overline{f}|$) such that $f_i = \mathcal{E}[\,\texttt{insync}\ l\,e\,]$ for some l, there is $\pi(l) = 1$.

Of course, a well-typed, closed expression with empty store is well-formed.

According to Lemma 10, the property $\vdash_{wf} \pi, \sigma \mid \overline{f}$ is maintained during evaluation.

Lemma 10 (Well-Formedness Preservation)**.** If $\vdash_{wf} \pi, \sigma \mid \overline{f}$ and $\pi, \sigma \mid \overline{f} \longrightarrow (\pi, \sigma)' \mid \overline{f}'$ then $\vdash_{wf} (\pi, \sigma)' \mid \overline{f}'$.

A state $\pi, \sigma \mid T$ is *deadlocked* if there exist only evaluation contexts $\mathcal{E}$, such that $T = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some verlocks l, such that $\pi(l) = 1$ for each l (i.e. the verlocks are not free) and there is no other evaluation context possible.

Now, we can state the progress theorem.

Theorem 6 (Progress)**.** Suppose T is a closed, well-typed term (that is, $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash T : t$ for some t and Σ). Then either T is a value or else, for any store π, σ such that $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash \pi, \sigma$ and $\vdash_{wf} \pi, \sigma \mid T$, there is some term T' and store $(\pi, \sigma)'$ with $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, or else T is deadlocked on some lock(s).

3.5 Related Work

There have been many proposals of concurrent languages with novel synchronization primitives, e.g. Concurrent Haskell [83], Concurrent ML [81], Polyphonic C# [6] and Pict [86]. They often enable to express complex synchronization code more easily than when using standard constructs, such as monitors and locks. Mobile agent languages based on concurrent process calculi, such as the join-calculus/JoCaml language [34, 30] and Nomadic Pict [97, 120, 111] offer, in addition, powerful means for synchronized, continuous communication between objects, which can freely move between machines and dynamically bind to local resources. This work is however orthogonal to the content of this book, in which we are primarily focused on high-level language support that can provide automatic concurrency control in modular protocol stacks.

The work in this chapter builds on research in three areas: atomic transactions, language support for atomicity, and formalization of isolation. Below we discuss example work in these areas.

3.5.1 Atomic transactions

Different forms of atomic transactions decomposed to satisfy individual transactional features, referred to by an acronym ACID (Atomicity, Consistency, Isolation, and Durability), have appeared in distributed operating systems (e.g. Camelot [29]), and in transactional platforms (e.g. Sun Enterprise JavaBeans (EJB) and Microsoft Transaction Server (MTS)). The main difference between the above systems and our work is that we design programming abstractions for atomicity of local code blocks that may have both data and I/O effects on a single machine; the design considerations and solutions were therefore different.

There have been a number of proposals of extending traditional programming languages with support of atomicity; we give some examples in the next section. A very flexible approach to composability of the ACID features can be found in Venari/ML [41, 109]. It is an extension of the ML programming language with atomic transactions. Higher-order functions in ML allow the programmer to easily compose transactional features. Transactions in Venari/ML can be multithreaded, similarly to our atomic tasks. Contrary to atomic transactions, however, our tasks never rollback their execution. We can therefore guarantee that any I/O operations are performed *exactly once* (unless, of course, a site with tasks crashes).

Alternative approaches to rollback in atomic transactions, such as *compensations* [16], i.e. implicit or programmable procedures that can undo the effects of a transaction that fails to complete, do not apply to network protocols. In general, the I/O operations performed by protocol tasks cannot be easily (or routinely) undone or compensated. In the protocol design, we usually assume that an output of a network message either succeeds and the message is sent, or the output fails (e.g. due to a socket error); in the latter case, the message is not sent. (Of course, a separate issue is if the message will be actually delivered.) The protocol designer should not be concerned with another case, when the message has been sent, but the operation needs some compensation due to conflicts on task operations.

3.5.2 Atomic code blocks

While our construct `atomic` can allow us to declare multithreaded sections of code to be executed in isolation, several researchers have proposed programming language features for atomicity of sequential actions executed by a thread. Below is the previous work closest to our own.

Flanagan and Qadeer [32] proposed a type system for specifying and verifying the atomicity of methods in multithreaded Java programs, where the notion of "atomicity" is similar to linearizability [46] for concurrent objects, and the notion of isolation in this book. Their approach allows program methods to be annotated with the keyword `atomic`. If the program type checks, then any interaction between an atomic method executed by a thread and steps of other threads is guaranteed to be benign, in the sense that these interactions do not change the program's overall behaviour. The type system is a synthesis of Lipton's theory of left and right movers (for proving properties of parallel programs) and type systems for race detection.

Our decision to allow tasks to be multithreaded means, however, that in our language it may not be possible to verify the isolation property statically (at compile time only), since the language allows task threads to be created and terminated dynamically at will. This, together with the requirements of rollback-freedom and language safety, motivates our hybrid, type-directed approach to concurrency control.

Moreover, applications that we consider may demand different levels of performance, isolation and real-time constraints; these varying demands will lead to a multiplicity of runtime concurrency controllers, based on a variety of scheduling algorithms (e.g. real-time algorithms [38]). Our intend is to allow the programmer to choose between different dynamic locking strategies, based on the available static information. Our declarative approach therefore differs from the above type-based approach to *verify* atomicity.

Harris and Fraser [44] have been investigating an extension of Java with (sequential only) atomic code blocks. Their proposal implements Hoare's conditional critical regions (CCRs) [48]. The programmer can guard a conditional region by an arbitrary boolean condition, with calling threads blocking until the guard is satisfied. It is also possible to explicitly terminate an execution of an atomic block and rollback, if some condition is not satisfied. The implementation is based on mapping CCRs onto a *Software Transactional Memory (STM)* [98], which groups together series of memory accesses and makes them appear atomic.

The main difference between their approach and ours is the lack of a need for rollback in the normal case of atomic execution. Unlike our pessimistic concurrency control, their implementation of atomicity depends on rollback and recovery. This restricts the availability of I/O operations within an atomic block. For instance, the STM-based implementation of atomic blocks in Haskell [45] forbids all operations that may have irrevocable I/O effects, which limits the scope of possible applications.

A plausible option could be based on buffering input operations (for possible recovery) and flushing all output operations on transaction commit (to prevent their duplication due to rollback). However, it does not seem to support an arbitrary pattern of I/O communication at real time.

3.5.3 Transaction models

Turning to the semantics of transactions, Chrysanthis and Ramamritham [22] have specified the broad spectrum of transactional models. More recently, Black *et al.* [13] have defined an equation theory of operators, where an operator corresponds to an individual ACID property. The operators can be composed, giving different semantics to transactions. The above models are however presented abstractly, without being integrated with any language or calculus.

Vitek *et al.* [106] and Jagannathan and Vitek [55] have proposed a calculi-based model of standard ACID transactions. They have formalized the optimistic and two-phase locking concurrency control strategies. Similarly to our approach, their formalization of the isolation property refers to the order (or scheduling) of concurrent actions. However, the soundness result rests upon an abstract notion of permutable actions, while our soundness result and proofs make explicit data accesses and task noninterference. This degree of detail allowed us to formally encode an example, version-based concurrency control algorithm. Moreover, the semantics of tasks and ACID transactions differs, as we have explained before.

Berger and Honda [8] have used a variant of π-calculus [74] to formalize the operational semantics of the standard two-phase commitment protocol for *distributed* transactions. This work however does not address local concurrency control (on a machine) and the isolation property.

Chapter 4

Declarative Synchronization

Developing multithreaded systems is considerably more difficult than implementing sequential programs due to several reasons: (1) Traditional concurrency constructs, such as monitors and conditional variables, are used to express synchronization constraints at the very low level of individual accesses to shared objects (thread safety). (2) Embedding the implementation of a synchronization policy in the main code compromises both a good understanding of the application logic, i.e. we are not sure from the first look what the application code does, and also an understanding of the policy expressed. (3) The notions of semantic roles such as producers and consumers, which are essential for the understanding of a given policy, tend to disappear beyond an accumulation of lines of code, just as the logical essence of a sequential program gets lost when expressed in, say, an assembly language. (4) Synchronization constructs are usually entangled with instructions of the main program, which means that the correctness of concurrent behaviour is bound to the correctness of the entire application; this feature complicates maintenance and code reuse—some of the most advocated reasons for using components.

Declarative synchronization assumes a clear separation of an object's functional behaviour and any synchronization constraints imposed on it. Such an approach enables the software developers to modify and customize synchronization policies constraining the execution of concurrency components, without changing the main code of these components. Thus, declarative synchronization supports code reuse, and makes programming easier and less error-prone. Moreover, adding a new component or replacing components in the concurrent system, which may require to revise synchronization policy, is not subjected to an inspection of the whole code of the system but only of declared synchronization constraints. Thus, declarative synchronization also supports unanticipated software evolution (see also Section 1.2.2). In this chapter, we describe two orthogonal approaches to declarative synchronization.

Firstly, we describe the calculus of *concurrency combinators*, allowing a program and its synchronization policy to be encoded separately. Synchronization policies include: true parallelism, evaluation order, and isolation. They are expressed abstractly using the concurrency combinators, i.e. compositional policy operators taking as arguments services and constraining the execution of the services at runtime. Separation of the synchronization and functional code gives, however, a way of expressing synchronization schemes that may not be satisfied by any program execution. A given program can only be executed for some range of synchronization schemes—the synchronization policies must be matched accordingly. Our calculus is therefore equipped with a type system, which is able to verify if the declared synchronization policy matches the program. Typable programs are guaranteed to satisfy the declared synchronization policy and make progress. A variant of the calculus presented in this chapter has been published in [112].

Then, we briefly describe a model of the *Role-based Synchronization (RBS)*, which assumes assigning semantic roles to concurrent threads, and using a declarative language for expressing constraints between the roles. The design of the RBS model has been guided by two main requirements: (1) to keep the semantics of synchronization control attached to the roles involved in the specification of a synchronization problem, rather than to the fragments of the component or object code; and (2) to allow expressing concurrent strategies independently from the code of the components (or objects), with some possibility to switch between different execution strategies on-the-fly. The RBS model has been introduced in [102]; the paper also describes an implementation of an example RBS synchronization package.

While some work on such *separation of concerns* exists (see [51, 35, 90, 89, 73] among others) and example language constructs have been proposed (see [88, 89, 72, 73]), to our best knowledge, the two approaches described in this chapter are distinct from similar languages (we characterize example differences in Section 4.6). We were also interested to address novel design problems like: When is it safe to spawn a new thread or call a method, so that the synchronization policy is not invalidated? Or, conversely, how can we build programs that are synchronization safe by construction? How should synchronization itself be implemented? What are new language features required? The work described in this chapter has provided new insights into this research area.

The chapter is organized as follows. Section 4.1 gives a small example. Section 4.2 introduces the calculus of concurrency combinators, describing its syntax and operational semantics. Section 4.3 shows an example program reduction. Section 4.4 describes the type system and main results. Section 4.5 describes the role-based synchronization model and the constraint language, and Section 4.6 contains related work.

4.1 Example Program

We begin with a small example. Below is a program, expressed using the call-by-value λ-calculus, extended with `let`-binders and references (defined in Chapter 3) from ML [75], and services. Services are expressions `Label #` e, where `Label` is a service name and e is the service expression (an implementation of the service). Execution of `Label #` e evaluates e to some value, and returns this value as the result of the whole expression. For simplicity, we omit types throughout the examples.

```
(* Declaration of data structures and functions *)

let r = ref 0
let a = λx. Update # (r := x; ())
let b = λy. Read # !r

(* The main expression *)

fork (a 1);
b ()    (* due to non-determinism, this call returns 0 or 1 *)
```

The above program creates a reference `r`, initialized to zero. Then, it defines two functions `a` and `b`. In the λ-calculus, functions are defined using `let`-expressions, by assigning a function abstraction to a name. Execution of the former function overwrites reference `r` with a value passed as the argument of the call, and returns a null value. Execution of the latter function returns the current content of reference `r`. (Recall that `!r` is the dereference operation, while `r := x` is the assignment operation.)

The bodies of functions `a` and `b` are attached to service names, respectively `Update` and `Read`; these names will be later used to define combinators. This approach to define services is rudimentary. In Chapter 6, we extend our calculus to an object-oriented language, in which object signatures (service names) can be dynamically bound/unbound to objects (service implementations).

In the main expression, the program spawns a new thread (using `fork`) that calls function `a`, in parallel with the main thread calling function `b`. Note that the threads (functions) share a reference cell `r`. Below we will refer to functions `a` and `b` via the corresponding service names. Thus, we could also say that in the main expression the `Update` and `Read` services are called in parallel. The object language in Chapter 6 will actually allow methods (functions) to be called via signature names.

The result returned by the program is the current content of the reference. Note that the execution of the program is non-deterministic: it may return either `0` or `1`, depending on thread interleaving. If the reference had been updated by service `Update` before it was read by service `Read`, the program returns `1`, otherwise the initial value `0` will be returned.

Synchronization policy

What if we want the most recent content of the reference to be returned by the program, but we still require that the services `Update` and `Read` are called by separate threads? For this, the main expression of the program could be extended with some synchronization code that synchronizes threads, so that the thread of service `Read` is blocked till service `Update` has completed. The synchronization code could be implemented using standard synchronization constructs. However, this approach does not support code reuse—a feature that may be demanded by modular systems (see Section 1.2.2). After the low-level synchronization constructs would have been entangled with the code of the above program, it could not be readily reused in another application. Let us propose a different solution.

Below we use the language of concurrency combinators to declare an example synchronization policy for our program; the policy declaration is separate from the main code of the program, which supports code reuse.

```
(* Declaration of combinators, data structures, and functions *)

Update foll Read    (* concurrency combinator *)

let r = ref 0
let a = λx. Update # (r := x; ())
let b = λy. Read # !r

(* The main expression *)

fork (a 1);
b ()    (* returns 1 in every run *)
```

Note that the code of the program has remained unchanged, except for adding declaration of a concurrency combinator `Update foll Read`, which should be read: "`Update` followed by `Read`". The combinator requires service expressions to be executed by two atomic tasks. However, the concurrent execution of these tasks must be equivalent to an ideal serial execution, in which the second task (executing service `Read`) commences after the first task has completed. Recall from Chapter 3 that the effects of dereference and assignment operations on references, executed as part of atomic tasks, are regarded to be the I/O effects of these tasks. Thus, the above program always returns the updated content of the reference, which we wanted.

Combinators satisfiability

The language of concurrency combinators introduces a new problem, however. Below our program has been modified, so that the declared synchronization policy *cannot* be satisfied by any execution of the program.

```
(* Declaration of combinators, data structures, and functions *)

Update foll Read     (* concurrency combinator *)

let r = ref 0
let a = λx. Update # (r := x; ())
let b = λy. Read # !r

(* The main expression *)

let z = b () in
a 1;     (* doesn't type check! *)
z
```

The synchronization policy is the same as before: `Update foll Read`. However, the two functions (services) are now called by the same thread, and in the order: `Read` is called first, then `Update`, which makes impossible to satisfy the synchronization policy declared. In other words, the executable code does not match the concurrency combinators declared, which means that the program is incorrect.

To verify such cases automatically, we have equipped our language of concurrency combinators with a type system able to verify satisfiability of combinators. Execution of a well-typed program is guaranteed to satisfy all combinators declared in the program. In other words, well-typed programs correctly implement the intended synchronization policy, declared using the combinators.

Typing for verification

The type system can be also used to verify if certain conditions hold in programs. Below we have modified our program, so that services `Update` and `Read` are called by the same thread. Nevertheless, combinator `Update foll Read` is satisfied, since the service call order in the program agrees with the order declared in the combinator.

```
(* Declaration of combinators, data structures, and functions *)

Update foll Read
Update || Read

let r = ref 0
let a = λx. Update # (r := x; ())
let b = λy. Read # !r

(* The main expression *)

b (a 1)      (* doesn't type check! *)
```

However, the program does not type check either. This is because we have also added a second combinator, `Update` $\|$ `Read`, which requires that the services `Update` and `Read` must be executed by separate threads (e.g. to support multiprocessor architectures). But this requirement is not satisfied in our program, since the service expressions are executed by the same thread of control. In order for the program to type check, we must therefore modify it, and, e.g. use another thread for the call of either function (as it was in the first variant of our program).

4.2 The Calculus of Concurrency Combinators

4.2.1 Syntax

The syntax of the *calculus of concurrency combinators* (or the *CK-calculus*, in short) is in Figure 4.1. The main syntactic categories are concurrency combinators and expressions. Below we describe all syntactic categories.

Services and composite services

Services are expressions of the form A `#` e, where A is the service name, and e is the service expression (an example implementation of service A). Service

Variables	x, y	$\in$ *Var*	
Service names	A, B, C	$\in$ *Mvar*	
Packages	p	$\in 2^{Mvar} \times 2^{Mvar}$	
Combinators	a, b, c		$::= A \mid a \parallel b \mid a \triangleright b \mid a\ \texttt{isol}\ b \mid a\ \texttt{foll}\ b$
Types	t		$::= \texttt{Unit} \mid t \rightarrow^p t$
Values	v, w	$\in$ *Val*	$::= () \mid \lambda^p x : t.\, e$
Declarations	K		$::= a \mid A = a \mid K\ K$
Expressions	e	$\in$ *Exp*	$::= x \mid v \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{fork}\ e \mid A\ \texttt{\#}\ e$
Programs	P		$::= (\mathrm{SP}, e)$

We work up to alpha-conversion of expressions throughout, with x binding in e in expressions $\lambda^p x : t.\, e$ and $\texttt{let}\, x = e'\ \texttt{in}\ e$.

Figure 4.1: The calculus of concurrency combinators: Syntax

names are ranged over by A, B, C. Execution of expression $A\ \texttt{\#}\ e$ evaluates e and returns the result of this evaluation. In a program, there can be many expressions of the form $A\ \texttt{\#}\ e$ (for a given service name A and any expression e), possibly evaluated by separate threads; all these expressions define jointly a single service A. We assume that the execution of service expressions is deadlock-free.

A *composite service* consists of services whose names have been somehow related by the concurrency combinators declared in the program; we call these services *subservices*. Below we confuse services and composite services unless stated otherwise. A service *completes* if the service expression returns a value. Execution of a composite service completes if all its subservices complete.

Concurrency combinators

Combinators are higher-order functions that contain no free variables. *Concurrency combinators*, denoted a, b, c, are combinators that can be used to specify synchronization policies. Formal parameters of these combinators are service names. We consider four basic concurrency combinators: $A \parallel B$, $A \triangleright B$, $A\ \texttt{isol}\ B$, and $A\ \texttt{foll}\ B$, where A and B are service names. They can be defined as follows:

1. The *parallel combinator* $A \parallel B$ requires services A and B to be executed by concurrent threads (e.g. to support multi-core processor architectures);

2. The *causal-order combinator* $A \triangleright B$ requires that service A commences execution before service B; however, if the services are not causally related then no constraints are specified, where the precise meaning of causality will be defined by the type system in Section 4.4;

3. The *isolation combinator* A `isol` B requires that the concurrent execution of services A and B must satisfy the isolation property; however, if the services are executed by the same thread, then no constraints are specified;

4. The *followed-by combinator* A `foll` B is similar to A `isol` B, but it requires that the concurrent (isolated) execution of services A and B that are causally related must be equivalent to an idealised, serial execution, in which service B commences execution after service A has completed; however, if services A and B are not causally related then A `foll` B is equivalent to A `isol` B; finally, if the services are executed by the same thread, then we only require service A to commence execution before B.

In our small calculus, service names can only be used to refer to service expressions in the definition of concurrency combinators. However, in case of the object-oriented language in Chapter 6, service names are object signatures that can be dynamically bound/unbound to objects (service implementations). The names of object signatures can then be used to call object methods or access object fields, without specifying the actual name of the object. Thus, the concurrency combinators for such a language could be used to define constraints between method calls or field accesses, made with the use of object signatures.

Combinator declarations

Combinator declarations K are collections of concurrency combinators. Combinators can be named, e.g. $A = a$ both declares combinator a and binds it to a fresh name A, which can be later applied in the definition of other combinators. In the concrete syntax of the language, parentheses could be used for grouping combinators.

Complex combinators A_0 op_0 ... op_{n-1} A_n are equivalent to a conjunction of n binary combinators A_i op_i A_{i+1}, where op_i ($i = 0..n-1$) is one of the combinator names ($\|$, $\triangleright$, `atomic`, or `foll`). For instance, the complex combinator $A \triangleright B$ `isol` C is equivalent to two binary combinators $A \triangleright B$ and B `isol` C, which ensure that service B can commence execution only after A has commenced its execution, and the execution of B is isolated from any concurrent execution of service C.

Types

Types include the base type `Unit` of unit expressions, and the type $t \rightarrow^p t$ of functions. It would be routine to add other types or subtyping on types to the calculus definition. To verify combinator satisfiability in function abstractions, the type of functions is decorated with a *service package* $p = (p_c, p_a)$, where p_c is a set of all service names which can be bound by a thread calling a function until the function returns, and p_a is the same as p_c but it also includes service names which can be bound by any threads spawned as the result of executing the function's expression. We assume that the information on services bound by functions can be provided explicitly, and leave type inference as an open problem.

Values

A value is either an empty value () of type `Unit`, or function abstraction $\lambda^p x : t.\, e$ (decorated with service package p). All values in the CK-calculus are first-class programming objects, i.e. they can be passed as arguments to functions and returned as results and stored in data structures.

Expressions

Basic expressions are mostly standard, including variables, values, function applications, and `let`-binders (we sometimes omit `in` in the `let` construct). The CK language allows multithreaded programs by including the expression `fork` e, which spawns a new thread for the evaluation of expression e. This evaluation is performed only for its effect; the result of e is never used. We use syntactic sugar $e_1; e_2$ for `let` $x = e_1$ `in` e_2 (for some x, where x is fresh).

We also assume the existence of standard ML-like references, but they are omitted here for simplicity. The operational semantics of references can be found in Chapter 3 (see Section 3.3.2). Recall that $!r$ is a dereference operation, and $:=$ is an assignment operation.

Programs

Synchronization policy is a set of all concurrency combinators of the form $A = B_1 \;\; op_1 \; ... \; op_{n-1} \;\; B_n$ and binary combinators $A \; op \; B$, extracted from combinator declaration K, where op_i $(i = 1..n-1)$ are combinator names (i.e. $\|$, $\triangleright$, `isol` or `foll`). For example, assuming that the following combinators have been declared: $C = A$ `foll` B and $C \;\triangleright D$, we have a synchronization policy SP $= \{C = A$ `foll` $B,\ A$ `foll` $B,\ C \;\triangleright D\}$.

A *program* is a pair (SP, e) of a synchronization policy SP and the program's expression e, where policy SP must be consistent, as defined below. We assume a fixed policy for each program.

4.2.2 Consistency of synchronization policy

To check consistency of synchronization policy, we must first identify constraints. A set of *constraints* CS can be constructed recursively from the synchronization policy SP, as follows. For each combinator binding $C = A_1\ op\ ...\ op'\ A_n$ in SP, replace every binary combinator c in SP of the form $X\ op\ Y$, where $X = C$ or $Y = C$, by n constraints c_i, such that c_i is exactly the same as c but the name C (on a selected position in c) is replaced by A_i ($i = 1..n$). For example, if $\mathrm{SP}_1 = \{C = A\ \texttt{foll}\ B,\ A\ \texttt{foll}\ B,\ C\ \triangleright D\}$, then $\mathrm{CS}_1 = \{A\ \texttt{foll}\ B,\ A\ \triangleright D,\ B\ \triangleright D\}$.

Synchronization policy SP is *consistent* if two conditions are met: (1) for any $A\ op\ B$ in CS ($A \neq B$), where *op* is an asymmetric operation (`foll` or $\triangleright$) and CS is a set of constraints constructed from SP, there is no $B\ op'\ A$ in CS, such that op' is also an asymmetric operation ($op' = op$ or $op' \neq op$), and (2) there is no $A\ op\ A$ in CS, where *op* is an asymmetric operation.

For example, consider a policy $\mathrm{SP}_2 = \{C = A \parallel B,\ A \parallel B,\ C\ \triangleright A\}$. Then, we have $\mathrm{CS}_2 = \{A \parallel B,\ A \triangleright A,\ B \triangleright A\}$. Thus, policy SP_2 is not consistent since $A\ \triangleright A$ is in CS_2, which is incorrect because $\triangleright$ is an asymmetric operation (the second condition does not hold). Similarly, policy $\mathrm{SP}_3 = \{C = A \parallel B,\ A \parallel B,\ C\ \ \triangleright C\}$ is also not consistent (both conditions do not hold). However, policy $\mathrm{SP}_4 = \{C = A \parallel B,\ A \parallel B,\ C\ \texttt{isol}\ C\}$ is consistent since `isol` is not asymmetric. Also, policy SP_1 is consistent.

4.2.3 Operational semantics

We specify the operational semantics using the rules defined in Figure 4.2 and 4.3. A state S is a collection of expressions, which are organized as a sequence $T_0, ..., T_n$, where each T_i in the sequence represents a *thread*. We use T, T' (with comma) to denote an unconstrained execution of threads T and T', and $T; T'$ (with semicolon) to denote that T' can commence only after T has reduced to a value, and (T) for grouping threads. We write $T \circ T'$ to mean either T, T' or $T; T'$.

The expressions f are written in the calculus presented in Section 4.2.1, extended with a new construct $A\{\,\tau\,\}$, which denotes a sequence of threads τ that is part of service A. The construct is not part of the language to be used by programmers; it is just to explain semantics.

State Space

$$
\begin{array}{rcll}
S & \in & \mathit{State} & = \quad \mathit{ThreadSeq} \\
T, \tau & \in & \mathit{ThreadSeq} & ::= \quad f \mid T,T' \mid T;T' \mid (T) \\
f & \in & \mathit{Exp}_{ext} & ::= \quad x \mid v \mid f\ e \mid v\ f \mid \texttt{let}\ x = f\ \texttt{in}\ e \mid \texttt{let}\ x = v\ \texttt{in}\ f \mid \\
 & & & \qquad \texttt{fork}\ e \mid A\ \texttt{\#}\ e \mid A\{\,\tau\,\}
\end{array}
$$

Evaluation and Service Contexts

$$
\begin{array}{rcl}
\mathcal{E} & = & [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ e \mid A\{\,\mathcal{E}\,\} \mid \mathcal{E},T \mid T,\mathcal{E} \mid \mathcal{E};T \mid v;\mathcal{E} \mid (\mathcal{E}) \\
\\
\mathcal{C} & = & [\,] \mid \overline{A\ op}\ \mathcal{C}\ \overline{op'\ A'} \qquad op \in \{\|,\ \triangleright,\ \texttt{atomic},\ \texttt{foll}\} \\
\\
A_{\mathcal{E}\,T}\{\,\tau\,\} & = & \mathcal{E}[\,A\{\,\tau\,\}\,];T \qquad \text{for some } T
\end{array}
$$

Structural Congruence Rules for Combinators

$$
\begin{array}{rcll}
a \parallel b & \equiv & b \parallel a & \text{(C-Prl)} \\
a\ \texttt{isol}\ b & \equiv & b\ \texttt{isol}\ a & \text{(C-Isol)}
\end{array}
$$

Structural Congruence Rules for Threads

$$
\begin{array}{rcll}
T,T' & \equiv & T',T & \text{(C-Sym)} \\
T \circ () & \equiv & T \qquad op \in \{\,,(\mathit{comma}),;(\mathit{semicolon})\} & \text{(C-Nil)} \\
();T & \equiv & T & \text{(C-Seq)}
\end{array}
$$

$$
\frac{T \longrightarrow T'}{\mathcal{E}[\,T\,] \longrightarrow \mathcal{E}[\,T'\,]} \quad \text{(C-Expr)}
$$

Figure 4.2: The CK-calculus: Reduction semantics – Part I

We define a small-step evaluation relation $e \longrightarrow e'$ read "expression e reduces to expression e' in one step". We also use $\longrightarrow^*$ for a sequence of small-step reductions, and a "meta" relation $\twoheadrightarrow$ (defined below) for many reduction steps with the isolation guarantee. Reductions are defined using evaluation context $\mathcal{E}$ for expressions and threads, and context $\mathcal{C}$ for synchronization policy rules. Context application is denoted by [], as in $\mathcal{E}[\,e\,]$. We write $\overline{A\ op}$ as shorthand for a possibly empty sequence $A\ op \ldots A'\ op'$ (and similarly for $\overline{op\ A}$).

We also use an abbreviation $A_{\mathcal{E}\,T}\{\,\tau\,\}$ for $\mathcal{E}[\,A\{\,\tau\,\}\,]; T$—i.e., "a context $\mathcal{E}$ of service A, followed by a (possibly empty) thread or a group of threads T that are blocked until A will complete". To lighten notation, we usually omit T in semantic rules, and write $A_{\mathcal{E}}\{\,\tau\,\}$.

Structural congruence rules are defined in Figure 4.2. They can be used to rewrite synchronization policy rules and thread expressions whenever needed.

The evaluation of a program (SP, e) starts in an initial state with a single thread that evaluates the program's expression e. Evaluation then takes place according to the transition rules in Figure 4.3. The rules specify the behaviour of the constructs of our calculus. The evaluation terminates once all threads have been reduced to values, in which case the value v of the initial, first thread T_0 is returned as the program's result. (A typing rule for `fork` will ensure that other values are empty values.) The execution of unconstrained threads can be arbitrarily interleaved. Since different interleavings may produce different results, the evaluator $eval(e, v)$ is therefore a relation, not a partial function. Below we describe the evaluation rules.

Basic evaluation rules

Nondeterministic choice (R-Choice) between states S and S', denoted $S + S'$, can lead to either S being evaluated and S' discarded, or opposite.

The next two evaluation rules are the standard rules of a call-by-value λ-calculus [87]. Function application $\lambda x.\, e\ v$ in rule (R-App) reduces to the function's body e in which a formal argument x is replaced with the actual argument v. The (R-Let) rule reduces $\texttt{let}\ x = v\ \texttt{in}\ e$ to the expression in which variable x is replaced by value v in e. We write $e\{v/x\}$ to denote the capture-free substitution of v for x in the expression e.

Service $A\ \texttt{\#}\ e$ in rule (R-Mark) marks the service expression e with the service name A; it reduces to the expression $A\{\,e\,\}$. The marking information will allow concurrency control rules (described below) to identify expressions that are part of a given service, and apply to them all relevant synchronization rules while evaluating these expressions.

Evaluator

$$\begin{array}{rcl} eval & \subseteq & Exp \times Val \\ eval(e, v) & \Leftrightarrow & e \longrightarrow^{*} T_0 \circ ... \circ T_n \quad \text{and} \quad T_0 = v,\ T_{j \neq 0} = () \end{array}$$

Transition Rules

$$S + S' \longrightarrow S \quad \text{or} \quad S + S' \longrightarrow S' \qquad \text{(R-Choice)}$$

$$\lambda x.\, e\ v \longrightarrow e\{v/x\} \qquad \text{(R-App)}$$

$$\texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow e\{v/x\} \qquad \text{(R-Let)}$$

$$A\ \texttt{\#}\ e \longrightarrow A\{\,e\,\} \qquad \text{(R-Mark)}$$

$$A\{\,v\,\} \longrightarrow v \qquad \text{(R-Compl)}$$

$$\mathcal{E}[\,\texttt{fork}\ e\,] \longrightarrow \mathcal{E}[\,()\,], e \qquad \text{(R-Fork)}$$

$$A\{\,\tau\,\}; T, A\{\,\tau'\,\}; T' \longrightarrow A\{\,\tau, \tau'\,\}; (T, T') \qquad \text{(R-Join)}$$

$$\frac{A = \mathcal{C}[\,B\ op\ \mathcal{C}'[\,C\,]\,] \in \text{SP} \qquad A \notin \mathcal{E}''}{\mathcal{E}''[\,B_{\mathcal{E}}\{\,\tau\,\} \circ C_{\mathcal{E}'}\{\,\tau'\,\}\,] \longrightarrow \mathcal{E}''[\,A\{\,B_{\mathcal{E}}\{\,\tau\,\} \circ C_{\mathcal{E}'}\{\,\tau'\,\}\,\}\,]} \qquad \text{(R-Fold)}$$

$$\frac{\begin{array}{c} A, B \text{ are the innermost services of redex} \\ A = B_1\ op\ ...\ op'\ B_n \in \text{SP} \qquad B \neq B_i \quad i = 1..n \end{array}}{A\{\,T \circ B_{\mathcal{E}}\{\,\tau\,\}, T'\,\} \longrightarrow A\{\,T, T'\,\} \circ B_{\mathcal{E}}\{\,\tau\,\}} \qquad \text{(R-Unfold)}$$

$$\frac{\begin{array}{c} A\ \texttt{foll}\ B \in \text{SP} \\ \mathcal{E}''[\,A_{\mathcal{E}}\{\,\tau\,\}; B_{\mathcal{E}'}\{\,\tau'\,\}\,] \longrightarrow^{*} S \end{array}}{\mathcal{E}''[\,A_{\mathcal{E}}\{\,\tau\,\}, B_{\mathcal{E}'}\{\,\tau'\,\}\,] \twoheadrightarrow S} \qquad \text{(R-Foll)}$$

$$\frac{\begin{array}{c} A\ \texttt{isol}\ B \in \text{SP} \\ \mathcal{E}''[\,A_{\mathcal{E}}\{\,\tau\,\}; B_{\mathcal{E}'}\{\,\tau'\,\}\,] \longrightarrow^{*} S \\ \mathcal{E}''[\,B_{\mathcal{E}'}\{\,\tau'\,\}; A_{\mathcal{E}}\{\,\tau\,\}\,] \longrightarrow^{*} S' \end{array}}{\mathcal{E}''[\,A_{\mathcal{E}}\{\,\tau\,\}, B_{\mathcal{E}'}\{\,\tau'\,\}\,] \twoheadrightarrow S + S'} \qquad \text{(R-Isol)}$$

Figure 4.3: The CK-calculus: Reduction semantics – Part II

The mark $A\{\,e\,\}$ will be erased when expression e evaluates to a value v, as defined by rule (R-Compl). Then, we say that service A has *completed*.

Evaluation of expression `fork` e creates a new thread which evaluates e, as defined by rule (R-Fork). A value returned by expression e will be discarded. If expression `fork` e is part of some service A, and we would like expression e to be part of this service after the new thread has been spawned, then e should be an expression A `#` e' for some e'.

The evaluation rules at the bottom half of Figure 4.3, beginning from (R-Join), define the semantics of concurrency control. Programs evaluated using these rules must be first type checked if the evaluation rules can be actually applied for a given synchronization policy. In Section 4.4, we present a type system that can verify it.

Folding and unfolding

The first rule, (R-Join), groups two concurrent expressions of the same service. The rule (R-Fold) encloses two concurrent services being part of a composite service A (as specified by the synchronization policy) with the name A. The rule (R-Unfold) removes service B (together with any threads blocked on B) outside the scope of a composite service A whose B is *not* part of, according to policy SP (otherwise applying the (R-Unfold) rule is forbidden). The abbreviations $A_{\mathcal{E}}\{\,\tau\,\}$ and $A_{\mathcal{E}'}\{\,\tau'\,\}$ allow contexts $\mathcal{E}$ and $\mathcal{E}'$ to be multithreaded, if needed by the reduced expressions.

Concurrency control

To explain remaining rules, we need to introduce a few definitions.

By *concurrent evaluation*, we mean a sequence of small-step reductions in which the reduction steps can be taken by threads with possible interleaving. Two services (possibly multithreaded) are executed *serially* if one service commences after another one has completed.

A *result* of evaluating a service A is any state S, such that $S \neq \mathcal{E}[\,A\{\,..\,\}\,]$ for any context $\mathcal{E}$. Note that states subsume the content of reference cells, represented with stores. An *effect* is any change to the content of stores. (For clarity, we have omitted the reference stores in the evaluation rules in Figure 4.3, but see Chapter 3 for a full exposition.)

We define *isolation* to mean that the effects of one service are not visible to other services executing concurrently—from the perspective of a service, it appears that services execute serially rather than in parallel.

The operational semantics of combinators A `isol` B and A `foll` B is captured using rules, respectively (R-Isol) and (R-Foll). The rules define the "isolated evaluation" relation $\twoheadrightarrow$, which specifies that the actual term containing services A and B (in the conclusion of each rule) should be evaluated by the small-step reduction ($\longrightarrow$) using all evaluation rules but (R-Isol) and (R-Foll). However, the order of applying the rules must be now constrained, so that *any* result S or S' of concurrent evaluation of the term, could be also obtained by evaluating a less concurrent term, given in the premises of rules (R-Isol) and (R-Foll), in which services A and B are executed serially.

The evaluation rules (R-Isol) and (R-Foll) are similar, except that the rule (R-Foll) restricts the ideal, serial execution to the execution in which B commences after A has completed, while the rule (R-Isol) does not specify the order in the ideal, serial execution; we omit details of the rules application. An implementation of "isolated evaluation" could use the versioning algorithms described in Chapters 2 and 3.

4.3 Example Reduction

Below is the evaluation of the program described in Section 4.1, according to the reduction rules of our calculus, and assuming a fixed synchronization policy SP $=$ {`Update foll Read`}. We extend states S to a pair $\langle \sigma \mid T \rangle$, where σ is the reference store, i.e. a finite map from reference locations to values stored in the references (see Section 3.3.2 for details). For brevity, we have used a concrete variable name `r` as the reference location; we have also omitted typing annotations.

```
< σ | let r = ref 0
      let a = λx. Update # (r := x; ())
      let b = λy. Read # !r
      fork (a 1);
      b () >

(R-Ref)
⟶ < (σ,r ↦ 0) |
      let a = λx. Update # (r := x; ())
      let b = λy. Read # !r
      fork (a 1);
      b () >

Syntactic sugar
⟶ < σ' |
```

```
      let a = λx. Update # (r := x; ())
      let b = λy. Read # !r
      let z = fork (a 1) in
      b () >

(R-Let)  (* Applied two times *)
⟶ < σ' |
      let z = fork (λx. Update # (r := x; ()) 1) in
      λy. Read # !r () >

(R-Fork)
⟶ < σ' |
      let z = () in
      λy. Read # !r (),
      λx. Update # (r := x; ()) 1 >

(R-Let)
⟶ < σ' |
      λy. Read # !r (),
      λx. Update # (r := x; ()) 1 >

(R-App)  (* Applied two times *)
⟶ < σ' |
      Read # !r,
      Update # (r := 1; ()) >

(R-Mark)  (* Applied two times *)
⟶ < σ' |
      Read { !r },
      Update { r := 1; () } >

(R-Foll)
↠  < σ' |
      Update { r := 1; () };
      Read { !r } >     where SP={Update foll Read}

(R-Assign)
↠  < σ'[r ↦ 1] |
      Update { (); () };
      Read { !r } >

Syntactic sugar
↠  < σ'' |
      Update { let x = () in () };
      Read { !r } >
```

```
(R-Let)
↠  < σ'' | Update { () }; Read { !r } >

(R-Compl)
↠  < σ'' | (); Read { !r } >      where σ''(r)=1

(R-Deref)
↠  < σ'' | (); Read { 1 } >

(R-Compl)
↠  < σ'' | (); 1 >

(C-Seq)
↠  < σ'' | 1 >
```

4.4 Type System

In this section, we present a type system, which is able to verify if synchronization policy declared using the concurrency combinators can be actually satisfied. Programs that cannot satisfy a given rule simply would not compile. The type system is not complete, i.e. some programs may not type check even if they correctly implement the synchronization policy.

4.4.1 Satisfiability of concurrency combinators

The semantics can be used to formalize the notion of *combinator satisfiability*, as follows. A thread T *binds* a service name A if there exists some evaluation context $\mathcal{E}$ such that $T = \mathcal{E}[\,A\{\,f\,\}\,]$ for some expression f.

A state S does *not satisfy* combinator $A \parallel B$ if its thread sequence contains a thread that binds both service names A and B. A state S does not satisfy combinator $A \;\triangleright B$ and also combinator A `foll` B, if its thread sequence contains either a term $\mathcal{E}[\,B\{\,f\,\}\,]$ such that $f = \mathcal{E}'[\,A \;\texttt{\#}\; e\,]$ (possibly $f = \mathcal{E}'[\texttt{fork}\;\mathcal{E}''[\,A \;\texttt{\#}\; e\,]\,]$), or a single- or multithreaded term $\mathcal{E}[\,B\{\,\tau\,\}\,];T$ and $T = \mathcal{E}'[\,A\{\,\tau'\,\}\,]$, for some contexts $\mathcal{E}$, $\mathcal{E}'$, $\mathcal{E}''$, expressions f, e, and thread sequences τ, τ'. In all other cases, the above combinators are *satisfied*.

Finally, combinator A `isol` B can be satisfied at runtime by all execution states. This is because any single-threaded evaluation of services A and B satisfies the combinator A `isol` B by definition. Otherwise, if A and B are evaluated by different threads, then rule (R-Isol) is applied to scheduling threads accordingly.

An execution run $S \longrightarrow^* S'$ does not satisfy a combinator a if it may yield a state that does not satisfy a, i.e. if there exists a state S'' in the run (including S and S') such that S'' does not satisfy a. Otherwise, we say that the run $S \longrightarrow^* S'$ *satisfies* combinator a.

4.4.2 Typing for combinator satisfiability

We define the type system using one judgment for expressions. The judgment and the static typing rules for reasoning about the judgment are given in Figures 4.4 and 4.5. The typing judgment has the form $\Gamma; \varkappa; p \vdash e : t$, read "expression e has type t in environment Γ, and is bound to service names $\varkappa$ of service package p", where an environment Γ is a finite mapping from free variables to types.

A *package* $p = (p_c, p_a)$ is defined by all service names which may be bound while evaluating expression e, either by the current thread only (p_c) or by all threads evaluating e (p_a); if e is single-threaded then $p_c = p_a$.

Our intend is that, given a policy SP if the judgment $\Gamma; \varkappa; p \vdash e : t$ holds, then expression e can satisfy all concurrency combinators in SP, and yields values of type t, provided the current thread has bound services described in $\varkappa$, it may bind at most services described in p, and the free variables of e are given bindings consistent with the typing environment Γ.

The core parts of typing rules for expressions are fairly straightforward and typical for the λ-calculus with threads evaluated only for their side-effects. The only unusual rule is (T-Mark); it type checks services of the form A `#` e, and requires the type of the whole expression to be e's type; it will be explained below.

The main feature of this type system is checking if expressions satisfy concurrency combinators. For this, we check if expressions do not invalidate any constraints CS imposed by the policy rules; see the definition of a constraints set CS in Section 4.2.2. The constraints are then used to define two kinds of relation between services: (A, B) *Prl* and (A, B) *Seq*. The relation (A, B) *Prl* (see Figure 4.5) declares services A and B to be *parallel services*, while relation (A, B) *Seq* declares services A and B to be *causally related*, or more precisely, it declares that service B *causally depends* on service A. The type system verifies if the relations declared are not contradicted by the program, as follows.

During typechecking, expressions are evaluated for a given constraint set CS, in the context of a package p and a set pair $\varkappa = (\varkappa_c, \varkappa_a)$ of *bound service names*, where $\varkappa_c$ are service names that have been bound by the current thread explicitly, using `#`, and $\varkappa_a$ are service names that are the same as in $\varkappa_c$ but

Judgments

$\Gamma; \varkappa; p \vdash e : t$ — e is a well-typed expression of type t in Γ, bound to service names in $\varkappa = (\varkappa_c, \varkappa_a)$ of service package $p = (p_c, p_a)$

Expression Typing

$$\frac{x : t \in \Gamma}{\Gamma; \varkappa; p \vdash x : t} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x : t; \emptyset; p \vdash e : t'}{\Gamma; \emptyset; \emptyset \vdash \lambda^p x : t.\, e : t \rightarrow^p t'} \qquad \text{(T-Abs)}$$

$$\frac{}{\Gamma; \varkappa; p \vdash () : \texttt{Unit}} \qquad \text{(T-Unit)}$$

$$\frac{\begin{array}{c}\Gamma; \varkappa \cup \{A\}; p \vdash e : t \\ A \in p_c \quad A \in p_a \\ \nexists B \in \varkappa_c\ (A, B)\ \mathit{Prl} \\ \nexists B \in \varkappa_a\ (A, B)\ \mathit{Seq}\end{array}}{\Gamma; \varkappa; p \vdash A\ \texttt{\#}\ e : t} \qquad \text{(T-Mark)}$$

$$\frac{\begin{array}{c}\Gamma; \varkappa; p \vdash e : t \\ (\Gamma, x : t); \varkappa \cup \varkappa'; p \vdash e' : t'\end{array}}{\Gamma; \varkappa \cup \varkappa'; p \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : t'} \qquad \text{(T-Let)}$$

$$\frac{\begin{array}{c}\Gamma; (\emptyset, \varkappa_a); (p'_c, p_a) \vdash e : t \\ p = (p_c, p_a) \quad p'_r \subseteq p_a\end{array}}{\Gamma; \varkappa; p \vdash \texttt{fork}\ e : \texttt{Unit}} \qquad \text{(T-Fork)}$$

$$\frac{\begin{array}{c}\Gamma; \varkappa; p \vdash e : t' \rightarrow^{p'} t \quad p' \subseteq p \\ \Gamma; \varkappa \cup \varkappa'; p \vdash e' : t' \\ \forall A \in p_c\ \nexists B \in \varkappa_c \cup \varkappa'_c\ (A, B)\ \mathit{Prl} \\ \forall A \in p_a\ \nexists B \in \varkappa_a \cup \varkappa'_a\ (A, B)\ \mathit{Seq}\end{array}}{\Gamma; \varkappa \cup \varkappa' \cup p'; p \vdash e\ e'\ : t} \qquad \text{(T-App)}$$

$x \cup p = (x_r \cup p, x_s \cup p)$

$x \subseteq x' \equiv x_r \subseteq x'_r$ and $x_s \subseteq x'_s$ where $x = \varkappa$ or $x = p$

Figure 4.4: The CK-calculus: Typing expressions – Part I

Auxiliary Definitions

$$\frac{A \parallel B \in \text{CS} \text{ or } B \parallel A \in \text{CS}}{(A,B)\ \mathit{Prl}} \text{ (T-Prl)} \qquad \frac{A \triangleright B \in \text{CS} \quad \text{or} \quad A\ \texttt{foll}\ B \in \text{CS}}{(A,B)\ \mathit{Seq}} \text{ (T-Seq)}$$

Figure 4.5: The CK-calculus: Typing expressions – Part II

also include names inherited by the current thread at spawning time, from a set $\varkappa_a$ of the parent thread.

A package $p = (p_c, p_a)$ decorates a function type and definition, representing all services that may be bound while evaluating the function by the current thread T only (p_c), and by T and also any other threads that are spawned as the effect of evaluating the function's body (p_a). For example, functions a and b in Section 4.1 have types, respectively $\texttt{Int} \rightarrow^{\{\texttt{Update,Update}\}} \texttt{Unit}$ and $\texttt{Unit} \rightarrow^{\{\texttt{Read,Read}\}} \texttt{Int}$.

The rule (T-Fork) requires the type of the whole expression to be `Unit`; this is correct since threads are evaluated only for their side effects. Note that the `fork`-ed expression is evaluated with $\varkappa_c$ nulled since verification of the $A \parallel B$ combinator requires that any spawned threads do not inherit service bindings from their parent thread (as we only check if A and B are not single-threaded).

Typing of service abstractions $A\ \texttt{\#}\ e$ is defined by the rule (T-Mark). Essentially, the rule checks two kinds of constraints: (1) if there is no service B in the set $\varkappa_c$ of services that could have been executed by the thread executing A so far, such that B had been declared to be parallel with A, and (2) if there is no service B in the set $\varkappa_a$ of services on which service A causally depends, such that B had been declared to causally depend on service A. If at least one such service B exists, then the program is not typable since the synchronization policy SP cannot be satisfied by any execution of the program. This is because either services A and B would not be executed by different threads as required by the parallelism relation $(A, B)\ \mathit{Prl}$, i.e. violating the true parallelism combinator $\parallel$, or services A and B would begin execution in the opposite order then required by the causality relation $(A, B)\ \mathit{Seq}$, i.e. violating either combinator $\triangleright$ or `foll`.

To support modular design of services, service abstractions $A\ \texttt{\#}\ e$ are typable only if they are inside a function abstraction; for this, we require in the (T-Mark) rule that A is in the package p.

The (T-App) rule defines typing of the function application. Essentially, the rule checks if for any service A in the package implemented by a function, there is no service B bound by a thread calling the function, which could contradict

Judgments

$$\vdash S : t \qquad S \text{ is a well-typed state of type } t$$

Rules

$$\frac{|T| > 0 \quad \Gamma; \emptyset; \emptyset \vdash T_i : t_i \quad \text{for all } \ i < |T|}{\vdash T : t_0} \text{(T-State)} \qquad \frac{\Gamma; \varkappa; p \vdash f_i : t_i \quad \Gamma; \varkappa'; p' \vdash f'_j : t_j \quad i < j}{\Gamma; \varkappa'; p' \vdash f_i \circ f'_j : t_i} \text{(T-Thread)}$$

$$\frac{\vdash S : t_0 \quad \vdash S' : t_0}{\vdash S + S' : t_0} \text{(T-Choice)} \qquad \frac{\Gamma; \varkappa; p \vdash c : t}{\Gamma; \varkappa; p \vdash A\{\, c \,\} : t} \text{(T-InService)}$$

Figure 4.6: Additional judgments and rules for typing states

any relations (A, B) *Prl* and (A, B) *Seq* that might have been declared by the synchronization policy; compare with typing of the A **#** e construct. If no such service B exists, then the function application is correctly typed.

The type system can be extended with reachability analysis of conditional branches and dead code elimination. Without this analysis, some defined policies may be rejected, even if all program executions would satisfy them.

4.4.3 Well-typed programs satisfy combinators

The fundamental property of the type system is that well-typed programs satisfy the declared synchronization policy, expressed using concurrency combinators. The first component of the proof of this property is a type preservation result stating that typing is preserved during evaluation. To prove this result, we extend typing judgments from expressions in *Exp* to states in *State* as shown in Figure 4.6. The judgment $\vdash S : t$ says that S is a well-typed state yielding values of type t.

Lemma 11 (Type Preservation)**.** *If $\Gamma; \varkappa; p \vdash S : t$ and $S \longrightarrow S'$, then $\Gamma; \varkappa; p \vdash S' : t$.*

Lemma 12 states that a program typable for some synchronization policy SP is reducible to states that satisfy all combinators in SP.

Lemma 12 (Combinator Preservation)**.** *Suppose $\Gamma; \emptyset; \emptyset \vdash S : t$ for some synchronization policy SP. If $S \longrightarrow^* S'$, then run $S \longrightarrow^* S'$ satisfies all combinators in SP up to state S'.*

Type preservation and combinator preservation "up to a state" ensure that if we start with a typable expression for some policy SP, then we cannot reach an untypable expression through any sequence of reductions, and the reduced expression satisfies combinators in SP. This by itself, however, does not yield type soundness. Lemma 13 states that evaluation of a typable expression cannot *get stuck*, i.e. either the expression is a value or there is some reduction defined.

Lemma 13 (Progress)**.** Suppose S is a closed, well-typed state (that is, $\vdash S : t$ for some t and policy SP). Then either S is a value or else, there is some state S' with $S \longrightarrow S'$.

We conclude that for a given policy SP, well-typed programs satisfy combinators in SP. An expression e is a *well-typed program* if it is closed and it has a type t in the empty type environment, written $\vdash e : t$.

Theorem 7 (Combinator Satisfiability)**.** Given a policy SP, if $\vdash e : t$, then all runs $e \longrightarrow^* v_0$, where v_0 is some value, satisfy combinators in SP.

4.5 Role-based Synchronization (RBS)

In this section, we describe the *Role-based Synchronization (RBS)*—another approach to declarative synchronization. The key idea is to abstract away from the concrete code of programs, and to express synchronization constraints at the level of program specification.

By looking at the classical synchronization problems, such as Producer-Consumer, Readers-Writers [23], and Dining Philosophers [26], we can identify two essential semantic categories which are used to describe these problems: roles and constraints imposed on the roles. Below we characterize these two categories.

4.5.1 Semantic roles

In synchronization problems, concurrent threads are often used to perform certain semantic roles, such as producers, consumers, readers, writers, and philosophers (in the dining philosophers problem). The definition of the semantic roles is part of the definition of a synchronization problem. Below we use the term *role*, meaning one or possibly many concurrent threads, logically representing the corresponding semantic role.

Roles can execute various *actions*, e.g. a consumer outputs a value to the buffer, a writer writes a value to a file, and a philosopher eats rice (or thinks). Roles can be in different *states* during program execution. Some actions are allowed only in certain states. Thus, in order to execute a given action, a role must first *enter* a state that allows the action to be executed, unless the role is already in such a state.

Many synchronization problems are concerned with accessing *shared resources* (or *shared objects*) in an exclusive manner. For example, a buffer is shared by producers and consumers, a file is shared by writers and readers, and forks are shared by philosophers. We can therefore identify two states: a state `In` of being able to call methods of a shared object (or execute some actions), and a state `Wait` of waiting to be able to do so. (The former state is denoted `In`, for being *in* a position to execute actions.) For instance, a producer is waiting if the buffer is full, a writer is waiting when another writer is writing, and a philosopher is waiting if at least one fork is missing. Otherwise, these roles are in the position to execute actions defined, correspondingly, on the buffer, the file, or the forks (and a rice bowl). Other states can be defined for more refined synchronization problems.

4.5.2 Synchronization constraints

Synchronization constraints [1] define conditions on when roles are allowed to enter states and execute actions allowed by the states. For example, a consumer can enter state `In` and input a value from the buffer if and only if: (1) neither other producer nor consumer is currently accessing the buffer, and (2) there is actually some value in the buffer. Many concurrent readers can read a file in parallel if there is no writer writing to the file at the same time. A philosopher can eat rice only if two forks are available. Note that the synchronization constraints specify a synchronization problem. If an implementation obeys all the constraints defined, then it provides a correct solution to the synchronization problem. Failure to satisfy any constraint in accessing shared objects by roles may cause incorrect program behaviour.

Below we define synchronization policy and synchronization guards—two parts that jointly define constraints on entering a state by a role.

Synchronization policy

Synchronization policy [2] defines constraint on roles and their states. Essentially, it specifies when a role is permitted or forbidden to enter a given state in terms of selected roles and their states. Thus, we can have two kinds of

[1] The notion of constraints in the RBS model should not be confused with the constraints in the concurrency combinators language.

[2] The notion of synchronization policy in the RBS model should not be confused with the synchronization policy in the concurrency combinators language.

synchronization policies: permission and denial (or refusal). Intuitively, a *permission policy* describes what must happen in order to *permit* a role to enter a state, while a *denial policy* describes what *forbids* a role to enter a state. The synchronization policy is *satisfied* if the constraint holds, and it is *violated* otherwise. Below we will only specify permission policies.

Synchronization policy that often appears in concurrent programming is "mutual exclusion", which states that some roles cannot be simultaneously in the critical section, i.e. they cannot simultaneously access a shared object (i.e. to call an object method, access a data field, etc.). For example, a producer and a consumer cannot access a shared buffer at the same time, two writers are not allowed to simultaneously write to the same file, and two philosophers cannot share the same fork. More precisely, they are not allowed to be both in the same state `In` at any time.

Synchronization guards

Satisfying synchronization policy is the necessary but often not sufficient condition to solve a given synchronization problem. For instance, a consumer cannot input a value if there is no value in the buffer. Thus, we also need a *synchronization guard*, which specifies logical conditions on the state of objects requested by a role; the conditions must be satisfied in order to allow the role to call the object's methods. The RBS constraint language allows several different synchronization constraints to be defined for a role to enter a state. Synchronization guards will be also used to specify logical conditions on which synchronization constraint should be chosen for synchronization.

4.5.3 The RBS constraint language

Figure 4.7 shows the RBS constraint language for declaring synchronization constraints. We use abbreviation $[\overline{x}]$ to denote a list of elements $\overline{x} = x_1; \ldots; x_n$, where [] is the empty list. Below we describe the language constructs.

Shared objects, denoted o, are declared as lists of pairs $(a, [\overline{\mathsf{S}}])$, where each pair specifies that in order to be able to call a method a of object o, a role must be in one of the states given in list $[\overline{\mathsf{S}}]$. The empty list [] is used if the corresponding method can be called in *any* state.

Constraint declaration `enter`(r, S) declares *synchronization constraints* on entering a state S by a role r; the declaration is a choice of pairs (P, G), written $(\mathsf{P}, \mathsf{G}) \vee \ldots \vee (\mathsf{P}, \mathsf{G})$, where P is the synchronization policy (of the permission kind) regulating the switching of role r to state S, and G is a synchronization guard, specifying any logical conditions on when a role is

Constraint Language

Objects	o	$\in \mathit{Objects}$
Actions (methods)	a	$\in \mathit{Actions}$
States	S	$\in \mathit{States} = \{\texttt{In}, \texttt{Wait}, ...\}$
Families	F	$\in \mathit{Families}$
Thread roles	r	$\in \mathit{Roles}$
Declarations	D	$\in \mathit{Declarations}$
Constraints	K	$\in \mathit{Constraints}$
Synchronization policies	P	$\in \mathit{Policies}$
Synchronization guards	G	$\in \mathit{Guards}$
Logical conditions	c	$\in \{o.a : \texttt{Unit} \rightarrow \texttt{Boolean}\}$
Policy types	t	$\in \mathit{Types}$
Policy rules	U	$\in \mathit{Rules}$

$$
\begin{array}{lll}
o & ::= & [(a, [\overline{\mathsf{S}}]); ... ; (a, [\overline{\mathsf{S}}])] \\
\mathsf{F} & ::= & ([\overline{r}], [\overline{\mathsf{S}}], [\overline{\mathsf{D}}]) \\
\mathsf{D} & ::= & \texttt{enter}(r, \mathsf{S}) = \mathsf{K} \vee ... \vee \mathsf{K} \\
\mathsf{K} & ::= & (\mathsf{P}, \mathsf{G}) \\
\mathsf{P} & ::= & (t, [\mathsf{U}; ... ; \mathsf{U}]) \\
\mathsf{G} & ::= & c \wedge ... \wedge c \\
t & ::= & \texttt{All-Required} \mid \texttt{All-Excluded} \mid \texttt{Some-Required} \mid ... \\
\mathsf{U} & ::= & ([\overline{r}], \mathsf{S})
\end{array}
$$

Example Policy Types

$$\frac{\forall r' \in \overline{r}_i.\ r' \text{ in } \mathsf{S}_i \quad \text{for all} \quad i = 1..n}{(\texttt{All-Required}, [([\overline{r}_1], \mathsf{S}_1); ... ; ([\overline{r}_n], \mathsf{S}_n)]) \ \text{ satisfied}} \quad \text{(P-All-Required)}$$

$$\frac{\forall r' \in \overline{r}_i.\ r' \text{ not in } \mathsf{S}_i \quad \text{for all} \quad i = 1..n}{(\texttt{All-Excluded}, [([\overline{r}_1], \mathsf{S}_1); ... ; ([\overline{r}_n], \mathsf{S}_n)]) \ \text{ satisfied}} \quad \text{(P-All-Excluded)}$$

$$\frac{\exists r' \in \overline{r}_i.\ r' \text{ in } \mathsf{S}_i \quad \text{for all} \quad i = 1..n}{(\texttt{Some-Required}, [([\overline{r}_1], \mathsf{S}_1); ... ; ([\overline{r}_n], \mathsf{S}_n)]) \ \text{ satisfied}} \quad \text{(P-Some-Required)}$$

$$\frac{\exists r' \in \overline{r}_i.\ r' \text{ not in } \mathsf{S}_i \quad \text{for all} \quad i = 1..n}{(\texttt{Some-Excluded}, [([\overline{r}_1], \mathsf{S}_1); ... ; ([\overline{r}_n], \mathsf{S}_n)]) \ \text{ satisfied}} \quad \text{(P-Some-Excluded)}$$

Figure 4.7: The role-based constraint language

allowed to enter S, e.g. shared objects may need to meet certain criteria. The operational semantics of a constraint $\texttt{enter}(r, S) = (P, G)$ is that role r can enter state S only if policy P is satisfied and guard G evaluates to true.

Below we explain in turn the declarations of the synchronization policy and of the synchronization guard.

Synchronization policy P is defined as a policy type t, paired with a list L of *policy rules*, where a policy rule U is a pair $([\overline{r}], S)$ of a list of roles and a state. The precise definition of when synchronization policy P is satisfied depends on the policy type t.

Consider a constraint $\texttt{enter}(r, S)$ defined as (P, G), with the synchronization policy $P = (t, L)$. In Figure 4.7, we have defined four example policy types t: `All-Required`, `All-Excluded`, `Some-Required` and `Some-Excluded`. The rules given specify when policy of a given type is satisfied.

1. $(\texttt{All-Required}, L)$ is satisfied only if for each tuple $([\overline{r}_i], S_i)$ in L, all roles $\overline{r}_i$ are in state S_i; the empty list of roles means *all* roles.

2. $(\texttt{All-Excluded}, L)$ is satisfied only if for each tuple $([\overline{r}_i], S_i)$ in L, all roles $\overline{r}_i$ are *not* in state S_i; the empty list of roles means *all* roles.

3. $(\texttt{Some-Required}, L)$ is satisfied only if for each tuple $([\overline{r}_i], S_i)$ in L, at least one role in $\overline{r}_i$ is in state S_i; the empty list of roles means *any* role.

4. $(\texttt{Some-Excluded}, L)$ is satisfied only if for each tuple $([\overline{r}_i], S_i)$ in L, at least one role in $\overline{r}_i$ is *not* in state S_i; the empty list of roles means *any* role.

In Figure 4.8, we illustrate the above definitions using an example policy rule $([r1, r2], S)$, where a state S is a big circle, and roles r_1 and r_2 are small circles; the cases of policy violation are crossed with lines. Roles in states required by the policy type are depicted with a filled circle, other roles are depicted with an unfilled circle. Satisfying policy `All-Required` implies that policy `Some-Required` is also satisfied, while satisfying policy `All-Excluded` implies that policy `Some-Excluded` is satisfied. Moreover, satisfying policy `All-Required` means that policies `All-Excluded` and `Some-Excluded` are violated, and vice versa. Finally, satisfying policy `All-Excluded` means that policies `All-Required` and `Some-Required` are violated, and vice versa.

A *synchronization guard* G in a constraint $\texttt{enter}(r, S) = (P, G)$ is the conjunction of logical conditions $c \wedge ... \wedge c$, where a *logical condition* c is a boolean function with no arguments; the function returns **true** if it would make sense to access the shared object by role r in state S at that moment,

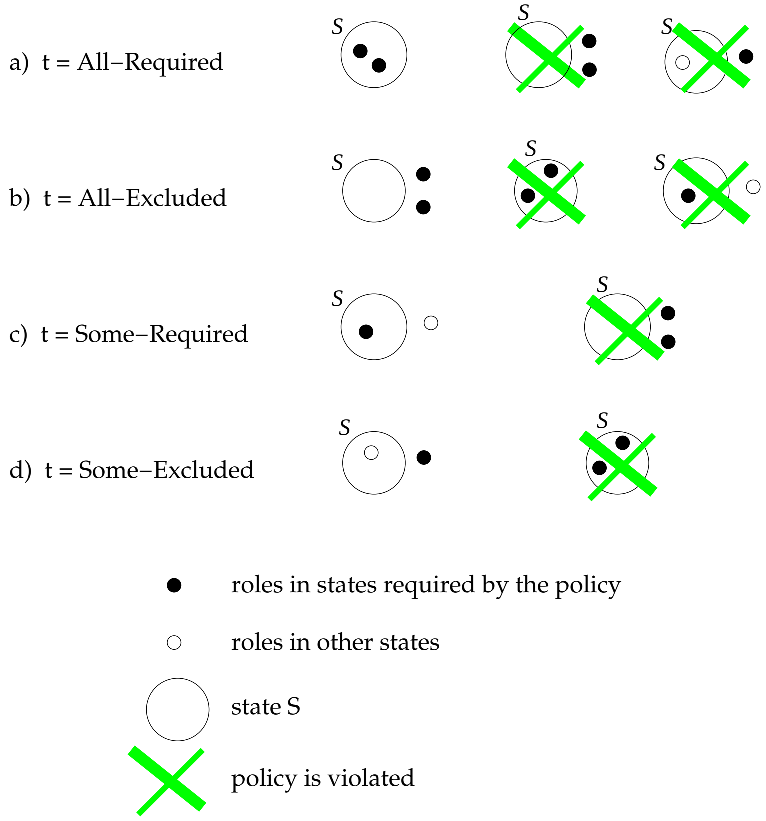


Figure 4.8: Examples of a policy $(t, ([r1, r2], S))$ satisfaction and violation

and **false** otherwise. What "makes sense" depends on the specification of the synchronization problem. Each function c in the guard is part of the program's code and can use any variables and data structures of the program. Thus, the state of the program can be used to express synchronization constraints.

To support dynamic change of synchronization policy, we allow a choice of synchronization constraints $(P_1, G_1) \vee \ldots \vee (P_n, G_n)$. Only a constraint whose guard G_i $(i = 1..n)$ has returned **true** is chosen for synchronization. If several guards have returned **true**, then the order of applying the corresponding policies is unspecified—it depends on the RBS synchronization package. The implementation of the RBS package should guarantee that transition between different synchronization policies is atomic.

Finally, we can define a *role family*, denoted F, to be a triple of a list of roles, a list of states, and a list of constraint declarations.

4.5.4 Example thread family

Consider a variant of the Producer-Consumer synchronization problem, specified as follows. Below we define a thread family `Producer-Consumer`, which declares two concurrent roles: producers and consumers. The roles are globally unique, and we assume that they share an object `Buffer`.

```
Producer-Consumer = ([Producer; Consumer], [In; Wait]
                     [enter(Producer, In); enter(Consumer, In)])

Buffer = [(output, [In]); (input, [In]); (is_empty, []);
          (is_full, [])]
```

An object `Buffer` has four public methods `output`, `input`, `is_empty`, and `is_full`; the former two methods can be called only by roles being in state `In`, while the latter method can be called by roles being in any state.

A producer can output some values to the buffer, and, in parallel, a consumer can input these values. Producers and consumers must have an exclusive access to the shared buffer; they must wait if the buffer is already accessed. Moreover, we require the priority of producers over consumers, i.e. a consumer is delayed not only if some producer outputs a value to the buffer, but also if some producer is waiting, even if the buffer would be free. We assume some notion of fairness, i.e. the delayed consumers and producers are eventually awakened when they are permitted to access the buffer.

Below we specify constraints for consumers and producers using a policy type `Excluded` and a synchronization guard with a single logical condition:

```
enter(Consumer, In) = (Excluded, [([], In), ([Producer], Wait)],
                       not buffer.is_empty())

enter(Producer, In) = (Excluded, [([], In)],
                       not buffer.is_full())
```

The synchronization constraint imposed on consumers for accessing the buffer `Buffer` is such that a consumer is forbidden to enter a state `In`, allowing the buffer to be accessed, if there is already a producer or consumer accessing the buffer or there are some producers waiting to access it. Moreover, a consumer can enter this state only when the buffer is not empty.

The constraint on producers for accessing the buffer `Buffer` is such that a producer is forbidden to enter state `In`, allowing the buffer to be accessed, if there is some consumer or producer accessing the buffer; moreover, it can enter state `In` only when the buffer is not full.

If synchronization policy would require a quantitative information about roles, we could add another field in a policy rule. This field would contain the minimal number of roles in a given state that makes the rule satisfiable (the default value is 1). For instance, replacing a policy rule `([Producer], Wait)` by a policy rule `(2, [Producer], Wait)` in the first constraint above, would mean that the consumer will be excluded only if there are at least two producers waiting. For clarity, we have omitted the quantitative information in the definition of the constraint language; however, the implementation of an example synchronization package in [102] supports this option.

Below we extend the above definition of the Producer-Consumer synchronization problem, and define two sets of synchronization constraints: the first one allows many consumers to access the buffer in parallel (not exclusively), and equals priorities of consumers and producers, while the second set of constraints is as before. The choice between the two sets of synchronization constraints is controlled in the synchronization guard by a boolean condition `equal_priority()`.

```
enter(Consumer, In) = (Excluded, [([Producer], In),
            equal_priority() ∧ not buffer.is_empty())

enter(Producer, In) = (Excluded, [([], In)],
            equal_priority() ∧ not buffer.is_full())

enter(Consumer, In) = (Excluded, [([], In),
                                  ([Producer], Wait)],
            not equal_priority() ∧ not buffer.is_empty())

enter(Producer, In) = (Excluded, [([], In)],
            not equal_priority() ∧ not buffer.is_full())
```

Since the sets of constraints are switched at runtime, depending on the value returned by a boolean function `equal_priority()`, the constraint declaration above demonstrates *dynamic switching* of synchronization policy—notably, the dynamic policy update does not require any modification to the code of the main program (implemented in the host language).

The simplicity and expressiveness of this formalism suggests that it can be indeed useful to encode synchronization constraints at the level of thread roles, instead of individual actions executed by the threads. The advantage is that the constraints can be expressed *declaratively*, as a set of policy rules. The rules are intuitively easier to understand than the low-level synchronization code, thus aiding design and proofs of correctness.

4.5.5 Implementation

To illustrate the role-based synchronization, an experimental implementation of the *Readers-Writers (RW)* synchronization package (design pattern) has been described in [102], where we also discussed an example application of RBS—the *Web Access* server with dynamic switching of access policy. The RW synchronization package has been implemented in the OCaml programming language [80]—an object-oriented variant of ML. OCaml has abstract types and pattern matching over types, which allowed us to have the concrete syntax of the constraint language almost exactly as in Section 4.5.3.

The key aspects of the RBS package implementation are as follows. Essentially, each call of an object method is preceded and followed by a call to a *synchronizer* object. The synchronizer is the core part of the synchronization package—it implements an evaluation engine parameterized over constraint declarations, expressed using abstract types in OCaml. The invocation of object methods by concurrent threads (roles) is guaranteed to satisfy any synchronization constraints imposed on the call, i.e. the call is suspended (and the thread blocked) if any constraint cannot be satisfied, and will be automatically resumed after actions made by other threads will allow the constraints to be satisfied.

4.6 Related Work

There have been a lot of work on separation of concerns in the broad area of software engineering (see [51, 63, 35, 90, 72, 73] among others). Below we discuss the related work in the area of separation of concurrency aspects and aspect-oriented programming; the issues of atomicity have been described in Chapters 2 and 3.

4.6.1 Separation of concurrency aspects

For a long time, the object-oriented community has been pointing out, under the term *inheritance anomaly* [67], that the concurrency control code interwoven with the code of classes can represent a serious obstacle to class inheritance. Milicia and Sassone [72, 73] addressed the inheritance anomaly problem, and proposed an extension of the Java programming language with a linear temporal logic for expressing synchronization constraints on object method calls. The language support of declarative synchronization, described in this chapter, is similar to their approach. However, our design has been motivated by the ease of programming and code reuse; we can also express complex synchronization policies such as atomicity. Our main focus was, however, to design a type system for combinator satisfiability.

Ramirez *et al.* [88, 89] have proposed a simple constraint logic language for expressing temporal constraints between "marked points" in concurrent programs. The approach has been demonstrated using Java, extended with the syntax for marking. Similarly to the work of Milicia and Sassone, the proposed language has however limited expressiveness. Contrary to the RBS constraint declarations that can freely access data and call boolean functions of a program, their constraints are not allowed to refer to program variables and data structures. Also, composite synchronization policies (on groups of threads) are not easily expressible.

The previous work, which set up goals similar to our own is also by Ren and Agha [90] on separation of an object's functional behaviour and the timing constraints imposed on it. They proposed an actor-based language for specifying and enforcing at runtime, real-time relations between events in a distributed system. Their work builds on the earlier work of Frølund and Agha [35] who developed language support for specifying multi-object coordination, expressed in the form of constraints, restricting invocations of a group of objects.

We are not aware of much work on formalizing combinator-like operations. Achermann and Nierstrasz [3] describe Piccola, which allows software components to be composed (although not isolated) using connectors, with rules governing their composition.

4.6.2 Aspect-oriented programming

An experimental implementation of the RBS synchronization package in OCaml resembles the weaving technique in the aspect-oriented programming.

The *Aspect-Oriented Programming (AOP)* approach is based on separately specifying the various *concerns* (or *aspects*) of a program and some description of their relationship, and then relying on the AOP tools to *weave* [54] or compose them together into a coherent program. For instance, error handling or security checks can be separated from a program's functional core. Hürsch and Lopes [51] identified various other concerns, including synchronization.

Lopes [63] developed a programming language D, which allowed thread synchronization to be expressed as a separate concern. Several AOP tools have been developed for popular programming languages. For example, AspectJ [56] allows aspect modules to be encoded using Java, and weaved at the intermediate level of the Java bytecode. The aspect code, written by the AspectJ programmer, is executed before and after the execution of pointcuts, where a *pointcut* usually corresponds to a method invocation. The code weaving technique could be used for encoding synchronization aspects, too. However, the pure AOP approach does not yet provide the expressiveness of a declarative synchronization language.

There are also differences in the implementation of the AOP approach and the RBS synchronization model. For example, translation of an AspectJ program with aspects into Java bytecode is done using a precompilation tool, while the RBS package implementation in OCaml, described in [102], uses entirely features of the host language. Thus, no external precompilation tool is required.

Chapter 5

Dynamic Protocol Update

In this chapter, we study *Dynamic Protocol Update (DPU)*, i.e. dynamic replacement of protocol modules across distributed machines. The DPU mechanism allows software modules or network components to be replaced on-the-fly without service interruption. The benefit is a decrease of software upgrade and maintenance costs in distributed systems that must run non-stop. Such type of dynamic software update introduces, however, a problem. To avoid interference between concurrent versions of the updated protocol, replacement of protocol modules must be (eventually) consistently performed on all machines, so that all services are provided correctly during and after the global update. This involves delicate synchronization (or coordination) of distributed updates which, if not handled appropriately, could easily prove so disruptive as to, at best, shut the system down, and, at worst, introduce malicious behaviour. Synchronizing updates on many sites, so that the whole system is updated in a consistent manner, and doing this while the system continuously provides service is a serious challenge (see Section 1.2.3). Moreover, the impact of any global synchronization should be minimized, so that system efficiency and scalability are not degraded.

Most of the existing work on programming language support for dynamic software update is concerned with local software upgrades or bug-fixes, assuming that the updated programs are essentially single-threaded and executed on a single machine, or, otherwise, giving no or weak guarantees that the system does not crash during or after update. There are quite a number of research papers that describe relevant implementations and techniques (see, e.g. [4, 15, 47, 66, 107, 28, 12, 101]). Most notably, Erlang [4]—a functional programming language designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications, offers support of code hot swapping. The code written in Erlang can be changed without stopping a system. A more recent work in the area of dynamic software update, concentrates on language support for *safe* code updating, i.e. with guarantees that the execution of compiled programs "cannot go wrong" when update occurs. Different approaches use static type systems that can guarantee type-safe runtime execution of updated programs (see, e.g. [66, 107, 47, 28, 12, 101]).

Some programming languages, such as Java, implement the mechanism of dynamic class loading [61] and linking [27], which can be used to plug-in whole software components at runtime; this mechanism simplifies the implementation of dynamically updateable systems. For example, there have been a lot of component and middleware systems developed using Java and other languages (see, e.g. [14, 57, 42, 100, 59] among others). These systems allow whole objects or components to be dynamically replaced, albeit often with no safety guarantees—the programmers must not forget to implement any missing synchronization of distributed updates, or exception handling methods to catch runtime errors. Also, the support of synchronization (or coordination) of distributed updates is not always satisfactory—e.g., few systems switch protocols in a fully distributed, decentralized way, i.e. without a central coordinator. Exceptions are, e.g., the Ensemble (Maestro) [105] toolkit for group communication, in which whole protocol stacks can be replaced at runtime in a coordinated manner [62], and the Cactus [21] protocol framework, that can be used to build adaptive systems, in which network components are switched dynamically using *barrier synchronization*.

The protocol frameworks, such as Maestro and Cactus, can be used to build dynamically updateable systems that are tolerant to crashes of individual machines. However, the support of DPU in these frameworks lacks simplicity and generality. We have therefore designed SAMOA [119, 92, 94]—a novel protocol framework, described briefly in Section 2.4, which currently provides the most flexible approach to dynamic protocol update. Individual protocols implemented in Java, using the SAMOA toolkit, can be seamlessly replaced on all sites by new protocols implementing the same service. Any synchronization required for the dynamic update is provided by the SAMOA switching algorithms; the algorithms must be designed independently for each updateable service, but otherwise they are transparent to the protocols being updated. The language features of Java, such as generics (polymorphic types), help to write type-correct code of updated components, with no uncaught errors. In Section 5.5, we briefly compare our approach with other similar toolkits.

In this chapter, we try to find answers to questions like what are the core properties of dynamic protocol update, and what is the range of protocol switching algorithms preserving these properties? For example, one of the critical safety properties of many network services that must be preserved by protocol switching algorithms is *message order*. Let us consider group communication middleware [95, 70] for replicating servers to make them tolerant to server crashes. Each server replica in the distributed group of servers is guaranteed to receive all messages in exactly the same order. Thus, any dynamic update of the middleware protocols must not change this semantics. On the other hand, some other network services may not require the message order property. Thus, we need a range of switching algorithms, each one used for updating of a different kind of service.

In order to understand what design choices can be considered, and what impact they have on the complexity and scalability of switching algorithms, we have defined a model of DPU. Then, we have used the model to present two synchronization-extreme switching strategies: a fully-synchronized algorithm that preserves message order but seems impractical for the Internet-wide update, and a synchronization-free, lazy algorithm that is scalable but does not guarantee the message order property. A preliminary description of the results described in this chapter appeared in [118].

The main results in a nutshell are as follows. Firstly, the lazy DPU strategy does not require any distributed infrastructure, which means that update with weak semantics is no more difficult than a local update. Secondly, the fully-synchronized and lazy strategies define the design space for more specialized switching algorithms that can use only as much synchrony as required in a given case. For illustration, in the end of the chapter, we summarize work on switching algorithms in SAMOA, designed and implemented for replacing distributed agreement protocols in the group communication stack (a complete description of this work appeared in [93, 91]). The switching algorithms preserve message order of agreement protocols but are more efficient than the fully-synchronized strategy.

The chapter is organized as follows. Section 5.1 defines our model. Section 5.2 describes properties of dynamic protocol update. Section 5.3 presents two example switching algorithms. Section 5.4 sketches the implementation work on dynamic update of distributed agreement protocols using the SAMOA toolkit, and Section 5.5 describes related work.

5.1 Model

Our model of DPU abstracts away from any concrete implementation of dynamic protocol update. We only make sure that the network communication is directly implementable above the Internet protocols. For this, we assume that

Symbols

Service names	A, B	$\in Mvar$
Protocol names	P, Q	$\in Mvar$
Required services	$\mathcal{R}$	$\in 2^{Mvar}$
Messages	m	
Protocol modules	a, b, c	
Module types	t	$::= (A, \mathcal{R})$ where $A \notin \mathcal{R}$
Module bindings	w	$::= \updownarrow \mid \uparrow$

Terms

Protocol stacks	$\mathcal{S} = \{a\ w, ..., b\ w'\}$
Module a in stack $\mathcal{S}$	$\mathcal{S}.a$
Protocol in ϕ encoded by a	$\Theta(a : t) = \{\mathcal{S}.a'\ w \mid \mathcal{S} \in \phi,\ a' : t\}$
Distributed systems	$\phi = \{\mathcal{S}_1, ..., \mathcal{S}_n\}$ or $\phi = \{\Theta(a), ..., \Theta(b)\}$
Messages sent to modules	$L_S = (m^{\mathcal{S}.a}, ...)$
Messages delivered by modules	$L_D = (m^{\mathcal{S}.a}, ...)$
Global message history	$L = (L_S,\ L_D)$
Global states	ϕ, L
A call of service A in $\mathcal{S}$	$\mathcal{S}.A(m)$
Delivery of m using $\mathcal{S}.a$	$\mathcal{S}.a \Uparrow m$

Figure 5.1: A model of DPU: Symbols and terms

protocols in our model use asynchronous, unordered, point-to-point messages; this is a realistic assumption about wide-area networks and common middleware services, where communication delays are not predictable. However, the model abstracts away from any unnecessary details of this communication. For instance, message addressing and message routing are the details of protocols themselves that we do not need to model here.

Below we define basic notions in our model of dynamic protocol update. All symbols and terms are in Figure 5.1.

5.1.1 Basic terms

Services provided by protocols are identified in our model using service names A, B. In Chapter 6, we define an object language in which service names are object signatures. In a service call, we specify a message that will be passed to the module implementing the service; messages are denoted by m.

Protocol modules (or *modules* in short), denoted by a, b, are software components encoding protocols, where each protocol provides some service. Modules are *typed* using pairs $(A, \mathcal{R})$, where A is the name of a service (interface) implemented by the module, and $\mathcal{R}$ are the names of all services that are required by the module to handle a call of A ($A \notin \mathcal{R}$). For instance, modules implementing protocols Q and P in Figure 5.4 have types, respectively $(B, \{A\})$ and $(A, \{\})$.

Protocol stacks (or *stacks* in short), denoted $\mathcal{S}$, are protocol modules composed together and located on a network site. In our model, we represent stacks by sets of modules, where each module is accompanied by the module's binding w, as in $\mathcal{S} = \{a\ w, ..., b\ w'\}$. Two kinds of module binding ($\updownarrow$ and $\uparrow$) will be explained in Section 5.1.5. We write $\mathcal{S}.a\ w$ to denote a module a in stack $\mathcal{S}$ with binding w. We often omit bindings or stack names if we mean any binding or any stack, or the stack is known from the context.

Protocols are sets of modules of the same type, i.e. each module of a given protocol must implement the same service (say A), and require the same services to handle the calls of A. We write $P = \Theta(a)$ to denote a protocol P consisting of modules of the same type as a; the modules in $\Theta(a)$ are accompanied by their bindings (see Figure 5.1). We normally assume that each module of a given protocol is in a different stack.

5.1.2 Distributed systems

A *distributed system*, denoted ϕ, can be defined in our model either vertically, as a set of stacks $\{\mathcal{S}_1, ..., \mathcal{S}_n\}$, or horizontally, as a set of protocols $\{P, ..., Q\}$. We say that each protocol in a protocol stack defines a level of abstraction (thus we say that ϕ is defined "horizontally" in terms of its protocols). For simplicity, we assume that any two stacks in the distributed system are exactly the same, unless the system is being updated. We also assume that there is at most one module of a given type in each protocol stack.

In the model we abstract away from physical machines—intuitively, stacks are located on machines (sites) that are interconnected via network. For clarity, we assume a system model with no failures, where messages are not lost nor duplicated, and stacks are basically reliable. In our implementation of DPU, however, protocol stacks may *crash* while the distributed system is being updated, with a guarantee that all non-crashed stacks get eventually updated.

Global states of a distributed system ϕ are tuples $\phi,\ L$, where $L = (L_S,\ L_D)$ is a pair of the (initially empty) lists of sent and delivered messages. We write $set(L_i)$ to denote a set of all elements in list L_i.

Communication

$$\frac{\begin{array}{c} a : (A, \mathcal{R}) \\ m^{\mathcal{S}.a} \notin set(L_S) \\ m^{\mathcal{S}.a} \notin set(L_D) \\ m^{\mathcal{S}.a} \in set(L'_S) \\ a\ w \in \mathcal{S} \quad w \in \{\updownarrow, \uparrow\} \quad \mathcal{S} \in \phi' \\ L''' = (L''_S,\ m^{\mathcal{S}.a} :: L''_D) \end{array}}{\phi,\ (L_S, L_D) \xrightarrow{A(m)} \phi,\ (L'_S, L_D) \longrightarrow^* \phi',\ (L''_S, L''_D) \xrightarrow{\mathcal{S}.a \Uparrow m} \phi',\ L'''} \text{(Comm)}$$

Free modules

$$\frac{L = (L_S,\ L_D) \quad L_S|a = L_D|a}{\phi,\ L \vdash a \text{ Free}} \text{(Freedom)}$$

where a global message history of ϕ, cut to a of type t:

$$L_S|a = \{m^{\mathcal{S}.a'} \mid m^{\mathcal{S}.a'} \in set(L_S),\ \mathcal{S} \in \phi,\ a' : t\}$$
$$L_D|a = \{m^{\mathcal{S}.a'} \mid m^{\mathcal{S}.a'} \in set(L_D),\ \mathcal{S} \in \phi,\ a' : t\}$$

Figure 5.2: A model of DPU: Communication and free modules

We model an *execution* (or *evaluation*) step of a distributed system ϕ using a state transition relation $\longrightarrow$, which transforms state ϕ, L to another state $(\phi,\ L)'$, as a result of a single action e, denoted $\xrightarrow{e}$; we sometimes omit the label e. The notation $(\phi,\ L)'$ means ϕ', L or ϕ, L' or ϕ', L', depending on the context. We also use $\longrightarrow^*$ to denote a possibly empty sequence of small step transitions.

5.1.3 Message passing

Protocol modules in different stacks communicate by means of asynchronous *messages*. We use two kinds of actions to express this communication: a service call and a message delivery.

Service call $\mathcal{S}.A(m)$ denotes a call of a service A in a stack $\mathcal{S}$ to deliver a fresh message m in another stack (or stacks if A is a multicast). As the result of this call a list of messages $(m^{\mathcal{S}'.a'}, ...)$ is appended to a global history L_S of *sent messages*, where each element in the appended list denotes message m decorated with a module that should deliver m.

For instance, we write $m^{\mathcal{S}'.a'}$ to denote that message m is going to be delivered in a stack S' by a module a'; we require that all modules to deliver message m must have the same type $(A, \mathcal{R})$ for some $\mathcal{R}$.

Message delivery $\mathcal{S}.a \Uparrow m$ denotes delivery of a message m in a stack $\mathcal{S}$ using a module a. The intended semantics is that message m is passed by a to another module in $\mathcal{S}$, that receives the message. We assume that the name of this module has been encoded in the message itself; we do not model this, however, as we only need to know who delivers a message. In the protocol frameworks, a message has a prefix, which is a list of headers, each header describing a protocol that should receive the message, e.g. a message $[a, m]$ should be received by $\Theta(a)$. As the result of message delivery $\mathcal{S}.a \Uparrow m$, an element $m^{\mathcal{S}.a}$ is added (using a Lisp-like constructor "::") to a global list L_D of *delivered messages*; list L_D has the same structure as L_S.

The (Comm) rule in Figure 5.2 specifies communication in our model: delivery $\mathcal{S}.a \Uparrow m$ of a message m by module $\mathcal{S}.a$ is always preceded by a call $A(m)$, where service A is implemented by the protocol $\Theta(a)$. There can be a number of evaluation steps in-between since the communication is asynchronous and protocols' (or an update's) actions can be interleaved. As the result of this communication, the sets of sent and delivered messages, respectively L_S and L_D, are modified accordingly (see Figure 5.2), where $L'_S = m^{\mathcal{S}.a} :: L_S$ for a point-to-point communication, and $L'_S = m^{s_1} :: ... :: m^{s_n} :: L_S$ for a multicast to n stacks ($s_i = \mathcal{S}.a$ for some i).

To deliver a message a protocol module in a stack usually depends on services at the lower level of abstraction; at the lowest level of abstraction the message is communicated to a destination stack. For example, a module a of type $(A, \mathcal{R})$, executed as the result of some call of service A, may call several services that a depends on—these services are in the set $\mathcal{R}$ of services required by a to handle a call of A. If $\mathcal{R} = \emptyset$ then the message is delivered to a module of the same type in another stack, where the message is passed upward the stack to the message recipient.

The details of message routing in a protocol stack and between stacks are omitted in our model, since they are not essential here. Example approaches can be found in [116, 117], where we describe the semantics of protocol composition and communication in the Cactus and Appia protocol frameworks.

5.1.4 Protocol rounds

An *execution round* of a protocol (or a *round* in short) is a sequence of reduction steps from a service call $\mathcal{S}.A(m)$ for some stack $\mathcal{S}$ and a fresh message m, until delivery of m for the last time. We normally assume that all rounds terminate. More precisely, a round begun with a fresh message m *terminates* (or *completes*) in a state ϕ, (L_S, L_D), if $L_S|m = L_D|m$, where $L_i|m$ ($i = S, D$) is a list constructed from list L_i by removing from it all messages other than

$$\frac{a\ w \in \mathcal{S} \quad w \neq w'}{a\ w' \notin \mathcal{S}} \qquad \text{(Binding-1)}$$

$$\frac{\begin{array}{c} a : (A, \mathcal{R}) \quad b : (A, \mathcal{R}') \quad a \neq b \\ a \updownarrow \in \mathcal{S} \quad b\ w \in \mathcal{S} \end{array}}{w = \uparrow} \qquad \text{(Binding-2)}$$

Figure 5.3: A model of DPU: Module bindings

m. There can be many rounds executed concurrently. A distributed system ϕ does not *get stuck* if all its rounds eventually complete.

A module a of a distributed system ϕ is *free* in a state ϕ, L, denoted $\phi, L \vdash a$ FREE, if in this state there are no uncompleted (non-terminated) rounds of any protocol that could deliver a message using either module a or other module (in any stack) that has the same type as a—i.e., there are no uncompleted (non-terminated) rounds of protocol $\Theta(a)$. We can formally define this property (called *freedom*) using global histories $L_i|a$ of messages that were sent ($i = S$) and delivered ($i = D$) by all modules of type of a; see rule (Freedom) in Figure 5.2.

5.1.5 Dynamic bindings

Modules can be added to, and removed from a protocol stack at runtime. For this, it should be possible to *bind* and *unbind* modules dynamically.

Consider a module a of type $(A, \mathcal{R})$ for some service A and $\mathcal{R}$. We write $a \updownarrow$ to denote that module a is *bound*, i.e. any calls of A in a stack can use a, and a can be also used to deliver messages in the stack. We write $a \uparrow$ to denote module a which is *passive*, i.e. any calls of A in a stack containing a cannot be made, unless there is another module b in the stack that also implements service A, and b is bound. A passive module can however deliver messages. Therefore, any round of a protocol $P = \Theta(a)$ can complete using any passive modules in P, if only any services required by P to complete the round in a given stack have bound modules in this stack.

Each module in a stack is either bound or passive. We also assume that for a given service A, there can be in each stack at most one bound module at a time that is implementing A. Both conditions have been defined formally in Figure 5.3.

5.2 Dynamic Protocol Update (DPU)

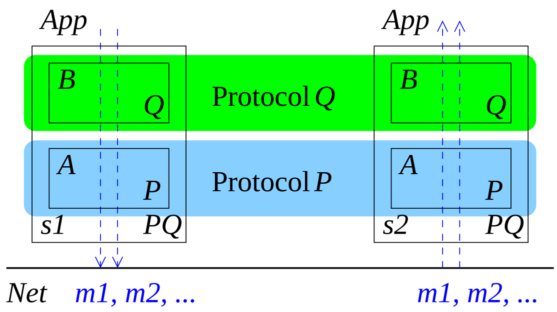
Dynamic Protocol Update (DPU) is any change of a distributed system ϕ, which means addition, removal or replacement of a protocol P in ϕ providing some service A. We require that the protocol update must *complete*, i.e. any addition, removal, or replacement of modules in P must eventually occur in all relevant stacks; after the addition or replacement operation has completed, any calls of service A will use the new protocol to deliver messages. In this section, we describe some fundamental properties of dynamic protocol update.

For example, Figure 5.4 shows a stack, in which a protocol P has been replaced by a new protocol N, both protocols providing the same service A. If N would require some nonexistent services, the implementations of these services should be added to every stack dynamically. The stack also contains another protocol (Q) that is above the protocol providing A. To communicate messages m_i $(i = 1, ..)$ to another stack, Q has made several calls of service A. Note that these calls are not interrupted by the fact that the implementation of A has changed in the middle, and all messages are delivered by Q in the destination stack. To guarantee this behaviour, some *switching algorithms* are required, that can coordinate (or synchronize) atomic update operations (such as a single module addition, removal, or replacement) with the concurrent execution of the updated distributed system. It is important to emphasize that the implementation of these algorithms is separate from the implementation of the protocols, i.e. they are part of the protocol framework's runtime system.

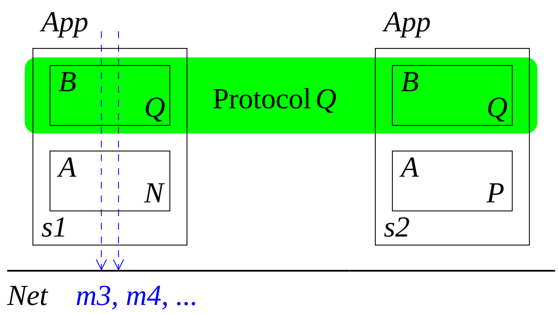
In practice, the design of switching algorithms is not trivial, e.g. they should deal with broadcast protocols, concurrent service calls, concurrent execution rounds of protocols, and concurrent updates, where all these actions may be initiated at different stacks. To make various design choices more clear, we define below the fundamental properties of switching algorithms for protocol replacement (or hot-swapping); the problems of protocol addition and removal are analogous, and they are not described here.

5.2.1 Replaceability

Static requirements about what can be replaced by what are twofold. Firstly, we can replace a module a by another module b in a stack $\mathcal{S}$ only if both modules implement the same service A. This requirement is backed up by the practice. If b would provide some service that is different from A, then the system updated with b could get stuck, since any calls of A made by other protocols that require A, could not be effectuated anymore. Secondly, we must also require that all services required by b are implemented in stack $\mathcal{S}$; if this a)



b)



c)

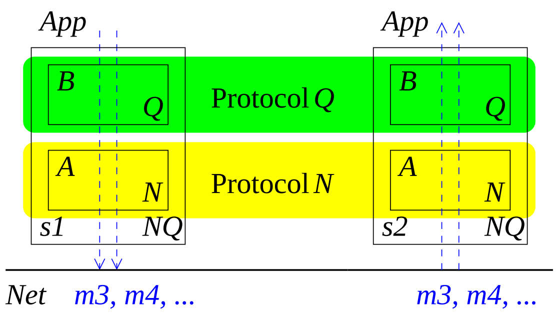


Figure 5.4: Replacement of P by N while Q communicates messages

$$\frac{\begin{array}{c} a : (A, \mathcal{R}) \quad a \in \mathcal{S} \\ b : (A, \mathcal{R}') \quad \mathcal{R}' = \{C \mid \exists c \in \mathcal{S},\ c : (C, \mathcal{R}'')\} \end{array}}{\mathcal{S}\{b/a\}} \qquad \text{(Replaceable)}$$

$$\frac{\phi = \{\mathcal{S}_1, ..., \mathcal{S}_n\} \quad \phi' = \{\mathcal{U}_L(\mathcal{S}_1, b/a), ..., \mathcal{U}_L(\mathcal{S}_n, b/a)\}}{\mathcal{U}_G : (\phi, b/a) \rightarrow \phi'} \qquad \text{(Global-Update)}$$

$$\frac{\mathcal{S}\{b/a\} \quad w \in \{\updownarrow, \uparrow\}}{\mathcal{U}_L : (\mathcal{S}, b/a) \rightarrow (\mathcal{S} \setminus \{a\, w\}) \cup \{a \uparrow\} \cup \{b \updownarrow\}} \qquad \text{(Local-Update)}$$

Figure 5.5: The DPU specifications

would not be the case, then the implementations of these services must be added to $\mathcal{S}$ before b.

Both requirements are specified by the (Replaceable) rule in Figure 5.5, which defines a *replaceability* property $\mathcal{S}\{b/a\}$, read "a is replaceable by b in stack $\mathcal{S}$". Let us consider a stack $\mathcal{S}$ that has a module a of type $(A, \{C\})$. Then, module a can be replaced in $\mathcal{S}$ by any module b whose type is $(A, \mathcal{R})$, where $\mathcal{R} = \{C\}$ or $\mathcal{R} \neq \{C\}$, if only $\mathcal{S}$ contains modules implementing all services in $\mathcal{R}$; we then write $\mathcal{S}\{b/a\}$. However, a cannot be replaced by any module of type $(B, \mathcal{R})$ for any B such that $B \neq A$.

Thus, verification of the replaceability property can be done statically (or when a protocol stack is built) by checking module types. If we had some notion of service subtyping, we could refine the (Replaceable) rule accordingly, and cover a larger class of modules that are replaceable one for another. Below we define what are the switching algorithms in our model.

5.2.2 Global update

A *global update* function $\mathcal{U}_G(\phi, b/a)$, defined by the (Global-Update) rule in Figure 5.5, updates all stacks of a distributed system ϕ, yielding an updated system ϕ' in which all occurrences of a module a in these stacks are replaced by a module b. To update a stack locally, $\mathcal{U}_G$ calls a local update function $\mathcal{U}_L$ (explained below). *Switching algorithms* are any coordination (or synchronization) algorithms implemented by the global update function $\mathcal{U}_G(\phi, b/a)$.

For practical reasons, whenever possible a global update of a distributed system should proceed concurrently with the execution of the system. Blocking the whole system during update is unrealistic for large systems, and also not acceptable for non-stop systems. Thus, the transition relation $\phi,\, L \overset{\mathcal{U}_G(\phi,b/a)}{\longrightarrow^*} \phi',\, L'$ transforming a system from a state before update to the state after the update has completed, consists of many evaluation steps, which may be interleaved (under control of the $\mathcal{U}_G$ algorithm) with other actions of the system being updated. We also allow several global updates to occur concurrently.

5.2.3 Local update

A *local update* function $\mathcal{U}_L$ (see the (Local-Update) rule in Figure 5.5) takes as arguments a stack $\mathcal{S}$, a module a in $\mathcal{S}$, and a new module b to replace a in $\mathcal{S}$, where the $\mathcal{S}\{b/a\}$ relation must hold, and yields a modified stack $\mathcal{S}$, in which b is bound and a is passive. This has an effect of replacing a by b in stack $\mathcal{S}$ in one atomic action. After a call of $\mathcal{U}_L$ returns, any calls of the service implemented by a and b will use its new implementation b instead of a. However, any pending rounds of protocols that need a can still complete since a is passive, which means that it can be used for delivering messages. In this abstract definition we do not specify when a can be safely removed from $\mathcal{S}$ (if ever); this depends on the switching algorithm.

The definition of global update does not specify *when* the function $\mathcal{U}_L$ is called. Updating some protocols at "wrong" moments may invalidate safety properties of these protocols. In Section 5.2.5, we identify two disjoint safety properties of dynamic protocol update: Message-Order and No-Message-Lost, that cover a broad range of distributed systems. Then, we describe in Section 5.3 two implementations of $\mathcal{U}_G$: the first one satisfies the former property, while the second one satisfies the latter one. However, before this we have yet to specify what do we mean by correctness of dynamic protocol update.

5.2.4 Correctness of dynamic protocol update

The replaceability property defined in Section 5.2.1 is necessary but not sufficient for correct dynamic protocol update. Below we give additional properties.

Consider global update $\mathcal{U}_G$ of a distributed system ϕ, which is dynamic replacement of a protocol providing some service A for another protocol that also provides A. We assume that both protocols are *correct*—i.e. they match the semantics of service A. Then, we can say (informally) that the global update $\mathcal{U}_G$ is transparent if it does not change the semantics of service A. In our model, the only observable effects of protocol execution that can be compared with the intended semantics of corresponding services are message

Judgements

$A \vdash_h L$	L is a correct global message history of service A
$A \vdash_{tgu} \mathcal{U}_G$	$\mathcal{U}_G$ is a transparent global update of service A

Message history

$$\frac{L = (nil, nil)}{A \vdash_h L} \qquad \text{(Null-History)}$$

$$\frac{\begin{array}{c} A \vdash_h (L'_S,\ L'_D) \\ A \vdash_h (L''_S,\ L''_D) \\ L = (L'_S@L''_S,\ L'_D@L''_D) \end{array}}{A \vdash_h L} \qquad \text{(Merged-History)}$$

Global update transparency

$$\frac{\begin{array}{c} \phi,\, L \xrightarrow{\mathcal{U}_G(\phi, b/a)}{}^{*} \phi',\, L' \\ a : (A, \mathcal{R}) \quad b : (A, \mathcal{R}') \\ A \vdash_h L \quad A \vdash_h L' \end{array}}{A \vdash_{tgu} \mathcal{U}_G} \qquad \text{(Transparency)}$$

Figure 5.6: Judgments, histories and transparency of DPU

outputs and message deliveries; both kinds of actions modify the system's state. Thus, we can define transparency more precisely as follows.

A global update $\mathcal{U}_G$ of a service A in a distributed system ϕ is *transparent*, if given any correct global message history L of service A, $\mathcal{U}_G$ transforms a system's state ϕ, L into a state ϕ', L', where global message history L' of A is also correct. In other words, the global message history of the updated service remains correct, despite of any dynamic replacement of the protocols implementing this service. Recall that a global message history L is a pair (L_S, L_D) of all messages that have been sent (L_S) and delivered (L_D) in ϕ, where each message is decorated with relevant names of modules and stacks. The meaning of message history correctness depends on the service semantics; we will discuss examples in Section 5.2.5.

We can define formally the transparency property using two judgments and the rules for reasoning about the judgments, as given in Figure 5.6. The judgment for message histories has the form $A \vdash_h L$, read "L is a correct global message history of service A". The judgment for global update $\mathcal{U}_G$ has the form $A \vdash_{tgu} \mathcal{U}_G$, read "$\mathcal{U}_G$ is a transparent global update of service A".

The rules (Null-History) and (Merged-History) are core rules for reasoning about message histories. They state that a message history constructed by appending two, possibly empty histories that are themselves correct is also correct. The Lisp-like append operation $L_i@L'_i$ returns a new list whose elements are the elements from the L_i and L'_i lists, in the order in which they appear in the corresponding lists.

Note that the transparency property (see the (Transparency) rule in Figure 5.6) essentially means that when no failures occurred, the service being updated remains operational during and after the update—i.e., any calls of this service made by other protocols are guaranteed to succeed, and the rounds caused by these calls are guaranteed to terminate. (Otherwise, the global message history would be incorrect.) In our model however we neglect any delays that may be caused by global update; good switching algorithms should minimize these delays.

In order to define transparency of global update, we have assumed that both an old protocol to be replaced as well as the new protocol replacing it, must correctly provide the same service. If the old protocol would be incorrect, then the global message history L just before the update commenced may not be correct either. Thus, any bug-fix update, i.e. replacement of an erroneous protocol by a correct one, is not transparent according to our definition. This agrees with the informal meaning of transparency, since the effect of such global update could be observed (e.g. errors are not repeated). On the other hand, replacement of a correct protocol by an incorrect one appears to be transparent till some errors manifest themselves. We do not define global update transparency in such a case.

Finally, we can say that a global update $\mathcal{U}_G$ of a service A in a distributed system ϕ is *correct* if it is transparent. This definition could be changed to accommodate more specific cases for which the above definition does not apply, e.g. replacement of an erroneous protocol for a correct one is not transparent.

5.2.5 Properties of dynamic protocol update

Our definition of global update transparency and correctness only requires that the global message history must match the semantics of the service being updated. Note that global update of services whose semantics permits messages to be lost, could in principle loose some messages, e.g. the User Datagram Protocol (UDP)—one of the core Internet protocols—does not provide the re-

$$\frac{\begin{array}{c}\phi,\, L \overset{\mathcal{U}_G(\phi,b/a)}{\longrightarrow^*} \phi',\, (L'_S,\ L'_D) \\ \phi,\, L \longrightarrow^* \phi,\, (L''_S,\ L''_D) \\ set(L'_S) = set(L''_S)\end{array}}{set(L'_D) = set(L''_D)} \quad \text{(No-Message-Lost)}$$

$$\frac{\begin{array}{c}\phi,\, L \overset{\mathcal{U}_G(\phi,b/a)}{\longrightarrow^*} \phi',\, L' \\ \phi,\, L \longrightarrow^* \overset{\mathcal{S}.b\Uparrow m}{\longrightarrow} \longrightarrow^* \phi'',\, L'' \\ L'' = (L''_S,\ L''_D)\end{array}}{L''_S|a = L''_D|a} \quad \text{(Message-Order)}$$

Figure 5.7: Properties of switching algorithms

liability and ordering that the Transmission Control Protocol (TCP) does, and so some messages could be lost during its update. Most applications however require that protocols should not loose messages. Some applications may also require that the protocols preserve certain order of message delivery.

We can therefore identify at least two classes of services in terms of message history properties: (i) services that do not loose nor duplicate messages, and (ii) services that guarantee certain order of message delivery. Correspondingly, we can define two properties of dynamic protocol update: No-Message-Lost and Message-Order, which are defined as follows.

Property 1 (No-Message-Lost)**.** A global update $\mathcal{U}_G$ of a distributed system ϕ has the *No-Message-Lost* property if: (i) $\mathcal{U}_G$ eventually terminates, and (ii) if ϕ does not get stuck, then the updated ϕ will deliver exactly the same set of messages as the non-updated ϕ would.

Switching algorithms that only satisfy the No-Message-Lost property cannot be used to update services ordering messages. Below is another property.

Property 2 (Message-Order)**.** A global update $\mathcal{U}_G$ of a distributed system ϕ, replacing one protocol for another one has the *Message-Order* property if: (i) $\mathcal{U}_G$ eventually terminates, and (ii) after a module of the new protocol has been used to deliver a message in some stack, the old protocol will never be used to deliver any message in any stack.

The above two properties have been concisely expressed using the (No-Message-Lost) and (Message-Order) rules in Figure 5.7, where global message histories $L_S|a$ and $L_D|a$, cut to a module a, were defined in Figure 5.2.

Consider dynamic update of some service that delivers messages in a certain order. The Message-Order property guarantees that a protocol of this service to be replaced by another protocol, can be used to deliver messages only until the new protocol will be used somewhere for the first time. Thus, up to this global time all messages sent by the old protocol are guaranteed to be delivered. Moreover, after this time no message can be sent using the old protocol. (Otherwise, condition $L''_S|a = L''_D|a$ in the (Message-Order) rule would not be true.) If both protocols are correct, i.e. their global message histories are correct and messages are delivered by these protocols with the prescribed order, then by (Merged-History) we have that the global history of the updated system is also correct.

Note that the No-Message-Lost property does not guarantee message order since it only guarantees that messages delivered during the global update are not lost, nor duplicated.

5.3 Switching Algorithms

Consider global update $\mathcal{U}_G$ of a distributed system ϕ, which means replacement of a protocol $\Theta(a)$ in ϕ by a new protocol $\Theta(b)$. Below we describe two example switching algorithms that could be used to implement $\mathcal{U}_G$.

We have defined the algorithms using a set of transition rules, each rule describing an atomic, single or double evaluation step of the algorithm. Atomic steps of the algorithms can be freely interleaved with the steps of other protocols in the system. The rules are expressed using the syntax in Figure 5.1, extended with polyadic messages (a *polyadic message* is a sequence of names or values). For readability, we give in each rule only part of the state, i.e. a local protocol stack in which a given action occurs, instead of a whole distributed system ϕ. In the description of the algorithms, we also use the notion of module bindings. Recall that bound modules (e.g. $a \updownarrow$) can be used both for service calls and message deliveries, while passive modules (e.g. $a \uparrow$) can only deliver messages in the stack.

Updating ϕ with a protocol $\Theta(b)$ may also involve adding some new protocols to each stack, so that all services required by $\Theta(b)$ are eventually provided. We omit an algorithm for adding new modules to the system; it could be similar to the second switching algorithm in this section.

5.3.1 Synchronized protocol update

The *Synchronized Dynamic Protocol Update (S-DPU)* algorithm updates a distributed system by replacing an old protocol in a distributed system ϕ by

$$\frac{\begin{array}{c} a, a' : (\mathrm{ABcast}, \mathcal{R}) \quad a \in \mathcal{S} \quad a' \in \mathcal{S}' \quad \mathcal{S}, \mathcal{S}' \in \phi \\ \phi,\, L \vdash a \ \textsc{Free} \quad L = (L_S, L_D) \end{array}}{L_D|\mathcal{S}.a = L_D|\mathcal{S}'.a'} \quad (\mathrm{ABcast})$$

Figure 5.8: The message delivery property of atomic broadcast

a new one. Firstly, it "passivates" bindings of the old and new modules in each stack, i.e. it sets up the bindings to $\uparrow$ so that the modules are passive. Then, the old module is removed and the new module is bound in every stack; this operation takes place locally only after it can be guaranteed that the old module will not be needed anymore to complete any protocol rounds.

To support concurrent global updates and termination of the global update under stack crashes, our switching algorithm communicates control messages using a *totally ordered broadcast* protocol [40], providing a service named *Atomic Broadcast (ABcast)*. We assume *abcast* to be some implementation of ABcast (see, e.g. a survey paper [24] for many examples of such protocols).

Execution of ABcast(m), where m is a fresh message, broadcasts m to all stacks with guarantees that the round of the $\Theta(abcast)$ protocol will terminate, and if some stack delivers message m before another message m', that was also broadcast using $\Theta(abcast)$, then every stack delivers m before m'. This property has been specified in our model using the (ABcast) rule in Figure 5.8, where the Free property is defined in Figure 5.2. The rule says that if we take any two protocol modules a and a' of type $(\mathrm{ABcast}, \mathcal{R})$ for some $\mathcal{R}$, that are in stacks, respectively $\mathcal{S}$ and $\mathcal{S}'$, then for any state ϕ, (L_S, L_D) of a distributed system ϕ, such that either module is free in this state, we have that the histories $L_D|\mathcal{S}.a$ and $L_D|\mathcal{S}'.a'$ of messages delivered by these modules up to this state are exactly the same.

Below are steps of the S-DPU algorithm (see Figure 5.9 for a precise definition). We assume that initially, i.e. when a message history is (nil, nil), all stacks in ϕ are identical.

Definition 8 (Synchronized Dynamic Protocol Update (S-DPU))**.**

S1: Broadcast a fresh message (S1, a, b) to all stacks, where a module a to be replaced by a module b is bound in a stack $\mathcal{S}$, and can be replaced in $\mathcal{S}$ by b.

S2: Upon receipt of (S1, a, b), passivate module a in the local stack $\mathcal{S}$ and extend $\mathcal{S}$ with passive module b. Then, broadcast a fresh message (S2, a, b).

$$\textit{S1}: \qquad \frac{\mathcal{S}\{b/a\} \quad a\updownarrow \in \mathcal{S}}{\mathcal{S},\, L \overset{\text{ABcast}(\text{S1},a,b)}{\longrightarrow} \mathcal{S},\, L}$$

$$\textit{S2}: \qquad \frac{\begin{array}{c}\mathcal{S}.\mathit{abcast} \Uparrow [\text{S1}, a, b] \quad \mathit{abcast} : (\text{ABcast}, ..) \\ \mathcal{S}' = (\mathcal{S} \setminus \{a\updownarrow\}) \cup \{a\uparrow\} \cup \{b\uparrow\}\end{array}}{\mathcal{S},\, L \longrightarrow \mathcal{S}',\, L \overset{\text{ABcast}(\text{S2},a,b)}{\longrightarrow} \mathcal{S}',\, L}$$

$$\textit{S3}: \qquad \frac{\begin{array}{c}\mathcal{S}.\mathit{abcast} \Uparrow [\text{S2}, a, b] \text{ from all } \mathcal{S}' \in \phi \\ \mathcal{S},\, L \vdash a \text{ I}\small{\text{DLE}}\end{array}}{\mathcal{S},\, L \overset{\text{ABcast}(\text{S3},a,b,\text{Finish})}{\longrightarrow} \mathcal{S},\, L}$$

$$\textit{S4}: \qquad \frac{\mathcal{S}.\mathit{abcast} \Uparrow [\text{S3}, a, b, \text{Finish}] \text{ from all } \mathcal{S}' \in \phi}{\mathcal{S},\, L \longrightarrow (\mathcal{S} \setminus \{a\uparrow\} \setminus \{b\uparrow\}) \cup \{b\updownarrow\},\, L}$$

$$\frac{L = (L_S,\ L_D) \quad L_S|\mathcal{S}.a = L_D|\mathcal{S}.a}{\mathcal{S},\, L \vdash a \text{ I}\small{\text{DLE}}} \qquad (\text{Idle})$$

$$\text{where} \quad \begin{array}{l} L_S|\mathcal{S}.a = \{m^{\mathcal{S}.a} \mid m^{\mathcal{S}.a} \in \mathit{set}(L_S)\} \\ L_D|\mathcal{S}.a = \{m^{\mathcal{S}.a} \mid m^{\mathcal{S}.a} \in \mathit{set}(L_D)\} \end{array}$$

Figure 5.9: The Synchronized DPU algorithm

S3: Upon receipt of (S2, a, b) from all stacks, wait until module a is idle in the local stack $\mathcal{S}$, then broadcast a fresh message (S3, a, b, Finish).

A module a of stack $\mathcal{S}$ is *idle*, denoted $\mathcal{S},\ L \vdash a$ IDLE where L is a message history, if all messages sent to a (by any stack) have been delivered by a [1].

S4: Upon receipt of (S3, a, b, Finish) from all stacks, remove module a from the local stack $\mathcal{S}$ and bind module b.

Note that the output and delivery of update-related control messages *do not modify message histories.*

Below is an example result about the S-DPU algorithm, which states that module rebinding by the algorithm is safe.

Lemma 14 (Safe Rebinding)**.** If S-DPU algorithm binds a new protocol module in some state, then the module being replaced with the new module is free in this state.

Proof. Consider binding of some module a in step *S4* of S-DPU. Then

1. by premise of *S4* and (ABcast), each stack $\mathcal{S}$ in ϕ has executed *S3*,
2. by 1. and *S3* and the definition of $\mathcal{S},\ L \vdash a$ IDLE, each stack $\mathcal{S}$ in ϕ has been in a state $\phi,\ L$, such that $L_S|\mathcal{S}.a = L_D|\mathcal{S}.a$,
3. by premise of *S3* and (ABcast), each stack has executed *S2*,
4. by 3. and *S2*, each stack has unbound a in *S3*, so $L_S|\mathcal{S}.a = L_D|\mathcal{S}.a$ is true not only in *S3* but also in *S4*,
5. by 4. and premise of *S4* and (ABcast), $L_S|\mathcal{S}.a = L_D|\mathcal{S}.a$ for all stacks $\mathcal{S} \in \phi$,
6. by 5. and the definition of $L_i|a$ $(i = S, D)$ in Figure 5.2, $L_S|a = L_D|a$,
7. by 6. and (Freedom), $\phi,\ L \vdash a$ FREE in *S4*,
8. by 7. and conclusion of *S4*, this completes the proof.

□

We conclude that the S-DPU algorithm satisfies the Message-Order.

Theorem 8 (S-DPU Safety)**.** Updating a distributed system with the S-DPU algorithm satisfies the Message-Order property.

[1] We assume the existence of a *global snapshot* algorithm [65] to determine this predicate.

Proof. Consider global update $\mathcal{U}_G(\phi, b/a)$. By Lemma 14 and *S4*, when a new module b is bound in a state ϕ, L, then an old module a is free. By *S2*, a has been unbound in state ϕ, L, i.e. for any state ϕ', L' following ϕ, L in *S4*, we have $L'_S|a = L'_D|a$. Since the new module b can be used for the first time to deliver a message in any state following state ϕ, L, by (Message-Order) this completes the proof. □

The S-DPU switching algorithm essentially implements a distributed hand-over of protocol modules in all stacks. To be able to replace modules safely, the system must be transformed to a global state which guarantees that no messages can ever be destined for the protocol to be replaced. This global predicate can be computed, e.g. using distributed algorithms for computing global states in asynchronous distributed systems (see, e.g. [65] for many examples of such algorithms). The S-DPU algorithm also depends on reliable broadcasts, preserving a total order of message delivery. Unfortunately, the totally ordered broadcast protocols that give such semantics are not efficient enough for some distributed applications.

Below we describe a switching algorithm that is efficient and well scalable but it does not satisfy the (Message-Order) property. It requires that each protocol message is communicated with a module that can be used to interpret the message; this module is installed in a stack upon message delivery (unless the stack has already installed the module). This means that the old and new implementations of a given service must coexist in every stack, since either of them may be used to deliver messages. In order to remove obsolete modules, some "garbage collection" algorithms could be designed; an example algorithm would be similar to the one computing the IDLE predicate.

5.3.2 Lazy protocol update

The *Lazy Dynamic Protocol Update (L-DPU)* algorithm updates a distributed system lazily, by extending stacks with a new module whenever needed.

We associate messages with modules that are used to deliver the messages. If a protocol module required to deliver a message is not in a local stack, then it is added to the stack, bound, and the binding of the old module implementing the same service is "passivated". Thus, any following rounds of the new protocol will use in this stack the new protocol module. However, any pending rounds of the old protocol can still complete using the "passivated" module. Therefore, the algorithm allows the old and new protocols to coexist in the distributed system for some time, i.e. they can deliver their messages concurrently.

L1: $b \notin \mathcal{S}$

$$\frac{\begin{array}{c}\mathcal{S}.c \Uparrow [b,m] \text{ for some } c \in \mathcal{S} \\ a \updownarrow \in \mathcal{S} \quad \mathcal{S}\{b/a\} \\ \mathcal{S}' = (\mathcal{S} \setminus \{a \updownarrow\}) \cup \{a \uparrow\} \cup \{b \updownarrow\} \\ L = (L_S,\ L_D)\end{array}}{\mathcal{S},\, L \longrightarrow \mathcal{S}',\, L \stackrel{\mathcal{S}.b \Uparrow m}{\longrightarrow} \mathcal{S}',\, (L_S,\ m^{\mathcal{S}.b} :: L_D)}$$

L2: $b \in \mathcal{S}$

$$\frac{\begin{array}{c}\mathcal{S}.c \Uparrow [b,m] \text{ for some } c \in \mathcal{S} \\ L = (L_S,\ L_D)\end{array}}{\mathcal{S},\, L \stackrel{\mathcal{S}.b \Uparrow m}{\longrightarrow} \mathcal{S},\, (L_S,\ m^{\mathcal{S}.b} :: L_D)}$$

Figure 5.10: The lazy DPU algorithm

The L-DPU algorithm does not require any distributed infrastructure, except the one used by the distributed system to communicate messages via network. Thus, it scales to large networks. Below are actions of the L-DPU algorithm (see Figure 5.10 for a definition in our model).

Definition 9 (Lazy Dynamic Protocol Update (L-DPU))**.**

L1: Upon delivery by a protocol module c in a stack $\mathcal{S}$ of a message $[b, m]$, if there is no module b in stack $\mathcal{S}$, then take any module a in $\mathcal{S}$ that is bound and replaceable by b, passivate a, and bind the module b from the message, so that the protocol $\Theta(b)$ can use message $[b, m]$. Finally, deliver m using b.

L2: Upon delivery by a protocol module c in a stack $\mathcal{S}$ of a message $[b, m]$, if module b is already in the stack $\mathcal{S}$, then the protocol $\Theta(b)$ can use message $[b, m]$. Finally, deliver m using b.

To guarantee termination of the global update, we could require that an "update" message containing any new modules is periodically broadcast to all stacks.

According to Theorem 9, the L-DPU algorithm guarantees that all protocol messages that have been sent are delivered but message ordering is not preserved.

Theorem 9 (L-DPU Safety)**.** Updating a distributed system with the L-DPU algorithm satisfies the No-Message-Lost property.

Proof. Straightforward by (No-Message-Lost) and atomicity of rebinding in *L1*. □

5.4 Example: Adaptive Group Communication

In [93, 91], we describe the *Adaptive Group Communication (AGC)* middleware, that has been developed to facilitate experimentation with switching algorithms for dynamic protocol update. The AGC middleware can be used to implement fault-tolerant distributed applications that must dynamically adapt to unanticipated changes in the network environment. The middleware protocols build on those from the Fortika group communication toolkit [71, 69, 33]; the AGC middleware has been implemented in Java, using the SAMOA protocol framework, described briefly in Section 2.

An unusual feature of our group communication middleware is that different protocols implementing the same service can be loaded, and switched dynamically. The switching occurs under control of specialized *switching algorithms*, which are designed for each updateable service independently from any existing or future protocols implementing these services. The switching algorithms guarantee global service availability and correctness while a distributed update operation takes place. They are executed by the SAMOA runtime system in background, transparently to the protocols replaced under their control. Therefore, the SAMOA programmers do not need to be aware that the protocols which they implement, can be loaded or replaced on-the-fly. The key idea of our design was to add a level of indirection between the service callers and the service provider; see [92] for details.

The main building blocks of the AGC system are *distributed agreement* protocols[1], providing essentially two services: *distributed consensus* and *atomic broadcast*. In [93, 91], we have described specialized switching algorithms developed for dynamically switching between arbitrary protocols implementing these services. Our switching algorithms maintain the Message-Order property, which is required by the distributed agreement. However, they are more efficient than the S-DPU algorithm described in this chapter. The key idea was to explore the semantics of the services to be updated, and in this way to avoid synchronization that was not necessary. In order to guarantee Message-Order,

[1] For information on example distributed agreement protocols and their specifications, see, e.g. [65, 19, 20, 95].

the switching algorithms in [93, 91] depend on the support of synchronization that is *already* provided by the distributed agreement services.

For example, let us consider global update of the distributed consensus service. This service ensures that given a group of distributed processes, after a round of consensus, all processes would agree on the same value, which has been chosen from values proposed individually by each process. Our switching algorithm has three steps, and uses the semantics of consensus for replacement of the consensus protocols, as follows. Firstly, an intend to replace a consensus protocol $\Theta(a)$ by $\Theta(b)$ is broadcast. Then, all processes must decide when b can be bound locally. For this, b could be piggybacked on any message that must be processed by the consensus service. Finally, when the decision about b has been delivered (that means *all* stacks reached consensus about binding b), a is passivated and b is bound; from now on, the new module b is used to process any subsequent messages. The time between binding a new module and making the old one passive is maximally reduced.

The details of the AGC switching algorithms and some results of performance measurements are in [93, 91]. The results show that the group communication service is available almost continuously while protocols implementing the updated distributed agreement service are switched on-the-fly. This practical experiment demonstrates that dynamic protocol update in a group communication system is feasible, and can be done efficiently. Most notably, our approach only uses the agreement service. This is in contrast to other adaptive group communication systems, which often require additional mechanisms, such as barrier synchronization (Graceful Adaptation [21]) or group membership (Maestro [105] and Appia [78]). Moreover, the application on top of the AGC middleware is never blocked, which is not the case in Maestro.

5.5 Related Work

The previous work closest to the work described in this chapter is on dynamic protocol update (DPU). While there has been a lot of interest in dynamic software updating (see, e.g. [47, 28, 4, 15, 66, 107, 12, 101, 14, 57, 42, 105, 21, 100, 59, 78] among others), and a number of implementations exist, relatively little work has been done on a rigorous design of dynamic update features for communication protocols. Below we discuss example protocol frameworks that support DPU. Next, we present selected work on formalization of DPU, and on language support for dynamic software updating (DSU).

5.5.1 Protocol frameworks for DPU

Recent years have seen a growing interest in the design of programming tools for *adaptive systems*, i.e. systems that can be reconfigured and adapted to new environments or changing user requirements; see [68] for examples of such tools and techniques. In particular, many programming tools and solutions have been proposed for dynamic protocol update [105, 21, 100, 59, 78]. However, some of these solutions (e.g. [100] and [59]) are clearly not satisfactory.

In [100], the authors propose a solution that uses a centralized manager, which limits its tolerance to failures. On the other hand, the solution proposed in [59] provides facilities to replace only a single module of a protocol, i.e. there is no synchronization (or coordination) of local and remote module replacements. Few systems offer any support for coordinating local updates. Contrary to SAMOA, described in Section 2.4, most of the systems that provide any coordination (or synchronization) of distributed updates, require an explicit interaction between the updateable protocols and the replacement manager, which leads to poor modularity since the implementation of DPU strongly depends on the updateable protocols. Thus, the designers of switching algorithms in these systems must understand the updateable protocols, while in our approach they only need to know the specification of updateable protocols. Below is the previous work closest to our own.

The Maestro [105] framework implements a switching protocol, which synchronizes dynamic replacement of protocol stacks in the Ensemble group communication toolkit. However, it only supports replacement of complete protocol stacks, i.e. in order to replace a single protocol, the whole stack (containing the protocol to be updated) is replaced, thus blocking, during update, all applications executed on top of the stack. Moreover, the design of protocols is not transparent to the dynamic update mechanism. In order to finalize the protocol stack, protocols in the stack must be extended with a method `finalize` that properly terminates the execution of these protocols. The `finalize` method is called by the *stack switch* module each time the stack replacement is required. An approach described in [78] is similar to Maestro but implemented using the Appia [76] protocol framework.

Graceful Adaptation [21] implements switching between network components using the Cactus protocol framework. A replacement manager, located on each host, interacts explicitly with replaceable network components. The local switch between components is done by deactivating the component that is currently active, and activating a new component. Each of these operations is performed by the components themselves; the replacement manager only coordinates all operations, thus the replacement operation is not transparent.

The replacement manager uses *barrier synchronization* for coordinating the beginning of the replacement across different hosts. A barrier for a group of threads or processes means that any thread (or process) must stop at this point and cannot proceed until all other threads (or processes) reach this barrier. The S-DPU switching algorithm, described in Section 5.3.1, resembles this idea. A similar solution has also been proposed in [100] but it uses a centralized manager, which limits its scope of applicability.

The approach to DPU in the SAMOA protocol framework has several advantages over above protocol frameworks. The main advantage is that the implementation of the switching module does not depend on any concrete implementation of the updateable protocols (and algorithms used by these protocols), but it is entirely based on the specification of the service implemented by these protocols. In Maestro and Graceful Adaptation, each updateable protocol module must be extended with code related to dynamic protocol update. To be able to write this code, the programmer must understand algorithmic details of the protocols that may be switched. Contrary to this, the switching algorithm in SAMOA is encoded entirely by the replacement module. Ordinary protocol modules are not even aware that the protocol replacement operation takes place. Our solution is therefore modular in contrast to existing solutions that require to extend *each* updateable module.

Another advantage of our approach is that it is highly flexible. Contrary to Graceful Adaptation, our solution does not limit possible replacements by imposing any restrictions on the services that the new protocol may require. Unlike Maestro, replacement of a single protocol in our system does not require a whole protocol stack to be replaced.

A more detailed comparison of the DPU support in Maestro, Grace Adaptation, and SAMOA, using adaptive group communication as an example, has appeared in [93].

5.5.2 Formalization of DPU

To date relatively little work has been carried out on formalization of dynamic protocol update. In particular, in none of the above systems (except for Maestro/Ensemble and SAMOA) is there any well developed evidence as to what conditions are needed to guarantee the correctness of updating distributed systems on-the-fly. Below we discuss this formalization work.

In [10, 62], the authors describe a generic switching protocol using the NUPR logical programming environment; the algorithm was intended for the Maestro/Ensemble [105] group communication toolkit. They have formally defined several meta properties on traces of send and deliver events, that should be preserved by updateable protocols. While in [118], we have identified space between lazy and synchronized update strategies with a correspondingly large number of very different switching algorithms, they only describe one switching algorithm. The algorithm is correct only for replacement of protocols that must exhibit all their (six) meta-properties; it cannot be applied for arbitrary protocols.

On the contrary, in the context of the SAMOA protocol framework, we have defined in [93], two generic correctness properties of dynamically updateable systems: *stack-well-formedness* and *protocol-operationability*. Preserving these properties and some additional correctness properties, which are specific to the protocols being replaced, guarantees that the dynamic protocol update is transparent to the users of the protocols.

5.5.3 Language support for DSU

Below we discuss example work in the area of language design for dynamic software update.

The Erlang [4] programming language allows software modules to be replaced at runtime, however with no safety guarantees.

The Java HotSpot Virtual Machine [50] allows a class instance to be replaced with the new instance in a running application through the debugger Application Programming Interfaces (APIs).

In the recent years there have been several efforts to support safe dynamic software updating by construction, i.e. to guarantee statically that the updated program remains type-correct. Below we discuss some of this work.

Dynamic ML [107] enables type-safe module replacement at runtime; changes can include the alternation of abstract types at update-time, and the addition (and possibly removal) of module definitions via garbage-collection. Dynamic Java classes [66] offer type safety preservation but compromise portability by modifying the Java Virtual Machine; also, class replacement is not synchronized with threads using old code. Duggan [28] describes a type-safe approach that allows a new module to change the types exported by the original module; it however does not discuss the rebinding facility.

Bierman *et al.* [11] study dynamic software updating with a small extension of a lambda calculus that supports an Erlang-like updating features. A preliminary discussion of safety properties is included. In continuation of this work, Stoyle *et al.* [101] investigate type-safe dynamic updating in C-like languages.

Methods of distributed versioning, such as Sewell's [96] fine-grain versioning control of values of abstract types, could be used to support interoperation of old and new modules.

Chapter 6

Dynamic Rebinding

To be able to compose and decompose software components at run time, some form of *dynamic rebinding* between components (or objects) is needed. In this chapter, we identify basic properties of dynamic object (re)binding, and define a class-based object calculus that gives precise meaning to these properties. A preliminary variant of this calculus has been published in [115].

What do we mean by *dynamic object rebinding*? Consider a construct `bind` A a that binds a name A to an object a. The effect of binding name A to a is that we can refer to a via name A, e.g. a method m of object a can be invoked either via $a.m$ or $A.m$. The crucial point here is that the object a can be later unbound from A (using a construct `unbind` A) and another object b can be rebound to A at runtime. By the alias change, any concurrent object c that knows name A, has been therefore unbound from a and bound to b.

We must ensure that types of objects a and b that are dynamically bound to A, match the corresponding types of fields and methods accessed or called via name A. For this, A is not a pure name but it is a *signature* that declares types of fields and methods of objects that are bindable to A. Objects are defined by classes, which define fields and methods with their types. Checking the match between signatures and classes is mostly standard; for clarity, we leave therefore our calculus untyped, focusing on the operational semantics. Note that an object c invoking a method $A.m$ may not even know the object on which method m is invoked. This simple mechanism can be used to implement software components (or objects with a predefined interface) that can be *composed dynamically*.

In this chapter, we investigate a small set of low-level language constructs that can be used to reason formally about dynamic object rebinding. In particular, we have used our language to give precise meaning to basic properties of dynamic object rebinding. We also define two example semantic properties that are often demanded by concurrent programs with low-level bind/unbind operations. We illustrate one of the semantic properties using a small example, erroneous program. Then, we show how the program can be fixed up using the synchronization abstractions designed in this book (the `atomic` construct and concurrency combinators).

The chapter is organized as follows. Section 6.1 introduces basic notions and defines the syntax of our calculus. Section 6.2 presents a set of language properties of dynamic object rebinding, and two example semantic properties of programs that use the dynamic rebinding feature. To illustrate one property, Section 6.3 shows an example erroneous program and how to fix it up. Section 6.4 formalizes the operational semantics of our language, thus giving precise meaning to the properties defined earlier. Finally, we discuss related work in Section 6.5.

6.1 The Object Calculus of Dynamic Rebinding

In this section we define the *calculus of dynamic rebinding* (or the *DR-calculus*, in short) as the call-by-value λ-calculus [87], extended with signatures, objects, object binding/unbinding, exceptions, threads and atomic tasks. The syntax of the language is in Figure 6.1. The main syntactic categories are signatures, classes, values and expressions. For convenience, we differentiate names: A, B range over signature names; P, Q range over class names; f ranges over object field names, and m ranges over method names. We write $\overline{x}$ as shorthand for a possibly empty sequence of variables $x_1, ..., x_n$ (and similarly for $\overline{t}$, $\overline{v}$, and $\overline{e}$). We abbreviate operations on pairs of sequences in the obvious way, writing e.g. $\overline{x} : \overline{t}$ as shorthand for $x_1 : t_1, ..., x_n : t_n$ (and similarly for $\overline{f = v}$). Sequences of parameter names in functions and class methods are assumed to contain no duplicate names. We write $\overline{M}$ as shorthand for a (non-empty) sequence of methods $M_1, \ldots, M_n$ in a class. Methods of the same class must contain no duplicate names; similarly, field names are unique per class.

6.1.1 Syntax

Types

Types include the base type `Unit` of unit expressions, which abstracts away from concrete ground types for basic constants (integers, Booleans, etc.), the type `Sig` of object signatures, the type `Obj` of objects, and the type $t \to t'$ of functions and class methods.

$$
\begin{array}{llcll}
\text{Variables} & x, y, a, b & \in & \mathit{Var} & \\
\text{Signature names} & A, B, C & \in & \mathit{Sig} & \\
\text{Class names} & P, Q & \in & \mathit{Lab} & \\
\text{Field names} & f & & & \\
\text{Method names} & m & & & \\
\text{Interface names} & n & \in & \mathit{Sel} & ::= \ f \mid m \\
\text{Types} & t & & & ::= \ \texttt{Unit} \mid \texttt{Sig} \mid \texttt{Obj} \mid \overline{t} \to t' \\
\text{Signatures} & s & & & ::= \ \texttt{sig}\ A\ \{f_1 : t_1, \ldots, f_k : t_k, \\
 & & & & \qquad m_1 : \overline{t}_1 \to t'_1, \ldots, m_n : \overline{t}_n \to t'_n\} \\
\text{Fun. abstractions} & F & & & ::= \ \overline{x} : \overline{t} = \{e\} \\
\text{Methods} & M & & & ::= \ t\ m\ F \\
\text{Classes} & \mathcal{C} & \in & \mathit{Class} & ::= \ \texttt{class}\ P\ \{f_1 = v_1, \ldots, f_k = v_k, \\
 & & & & \qquad M_1, \ldots, M_n\} \\
\text{Values} & v, w & \in & \mathit{Val} & ::= \ () \mid A \mid \texttt{new}\ P \mid F \\
\text{Expressions} & e & \in & \mathit{Exp} & ::= \ x \mid v \mid e.n \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & & & \quad \mid e := e \mid \texttt{bind}\ e\ e \mid \texttt{unbind}\ e \\
 & & & & \quad \mid \texttt{try}\ e\ \texttt{catch}\ e \mid \texttt{escape} \mid \texttt{fork}\ e
\end{array}
$$

We work up to alpha-conversion of expressions throughout, with $\overline{x}$ binding in e in an expression $\overline{x} : \overline{t} = \{e\}$, and x in e' in an expression $\texttt{let}\ x = e\ \texttt{in}\ e'$. Names do not bind, and so are not subject to alpha-conversion.

Figure 6.1: The class-based object calculus of dynamic rebinding: Syntax

Signatures

A *signature* describes an object interface, i.e. a declaration of object fields and methods that can be accessed or called upon an object via the signature. Syntactically, a signature is a keyword `sig`, followed by the name of the signature, and a sequence of field and method names, accompanied with their types. If an object implements some *service*, then the name of this service is the signature.

Methods

A *method* of the form $t\ m\ F$ has declarations of a type t of the values that it returns, its name m, and its body F. Access control is not modelled (all fields and methods are public). Objects can refer to their own methods with *self.m*, where *self* is a variable. A method's body is a function abstraction of the form $\overline{x} : \overline{t} = \{e\}$ (we adopted the C++ or Java notation, instead of the usual $\lambda\overline{x} : \overline{t}.e$ from the λ-calculus).

Classes

A *class* has declarations of its name (e.g. `class` P) and the class body $\{\overline{f} = \overline{v}, \overline{M}\}$, where $\overline{f} = \overline{v}$ is a sequence of fields (data containers) accessible via names $\overline{f}$ and instantiated to values $\overline{v}$, and $\overline{M}$ is a sequence of object methods. Classes do not explicitly declare their superclass with `extends` since we do not model class inheritance. Class inheritance and object constructor methods can be easily added to the calculus definition, in the style of Featherweight Java (FJ) [53]. We assume that every class implicitly extends a special class `Object`, like in FJ. The class `Object` does not define any fields nor methods.

Values

A *value* is either an empty value () of type `Unit`, a signature name, e.g. A, an object instance, e.g. `new` P, or function abstraction $\overline{x} : \overline{t} = \{e\}$. Values are first-class, they can be passed as arguments to functions and methods, and returned as results or extruded outside objects. (Typing could be used to forbid extruding functions that contain object *self* references.)

Basic expressions

Basic expressions e are mostly standard and include variables, values, field/method selectors, function/method applications, `let` binders, and field assignment $e := e$. The `let`-binder is a construct of ML-like languages, that can be used to define functions, and to bind object and immutable data to variables. For instance, `let` $x =$ `new` P `in` e creates a new object of class P that is bound to a variable x (where x binds in e). Then, we can write e.g. $x.f := v$ to overwrite a field f of object x with a value v, or we can write e.g. $x.m\ v$ to call a method m of object x. We use syntactic sugar $e_1; e_2$ (sequential execution) for `let` $x = e_1$ `in` e_2 (for some x, where x is fresh).

Dynamic binders and exceptions

Execution of `bind` $A\ a$ binds a signature A to an object a; any previous binding of signature A disappears. Execution of `unbind` A unbinds a signature A from any object bound to A, or raises an exception if no object is bound to A.

To catch exceptions, we have an expression `try` e `catch` e', which is similar to the one found in ML-like languages. If there was an exception thrown in e then the execution of e terminates and e' commences. Execution of `try` e `catch` e' returns either the result of e, if no exception occurred, or the result of e', if there was an exception thrown in e and no exception in e'. Exceptions can be thrown explicitly using `escape`, or implicitly (as in `unbind`). If there is no expression to catch an exception, the execution of `escape` blocks its thread of execution.

Threads

The language allows multithreaded programs by including an expression `fork` e, which spawns a new thread for the evaluation of expression e. This evaluation is performed only for its effect; the result of e is never used.

Programs

A *program* is a pair (CT, e) of a class table CT and a main expression e, where the class table CT is a mapping from class names to class declarations. To lighten the notation, we always assume a *fixed* class table CT. To avoid uncaught exceptions we syntactically restrict the program's main expression e to have the form `try` e' `catch` v, where v is a value. We assume that a class table satisfies some sanity conditions: (1) $\text{CT}(P) = \texttt{class}\ P \ldots$; (2) $\texttt{Object} \notin \text{CT}$; and (3) for every class name P (except `Object`) appearing anywhere in CT, we have $P \in \mathit{dom}(\text{CT})$. Given these conditions, a class table can be easily identified with a sequence of class declarations.

6.2 Properties of Dynamic Rebinding

Below we present basic properties of language constructs for binding/unbind objects in our calculus, together with some discussion of higher-level rebinding constructs that could be built on top of our calculus.

Then, we give two example semantic properties of *programs*, in which objects can be rebound dynamically. The untyped calculus presented in this chapter does not have language support to declare and verify if such semantic properties hold. We leave this for future work.

6.2.1 Language properties

Below are runtime properties of the language constructs. After each property, we provide a short justification of our design choice.

Property 3 (Binding Uniqueness)**.** At run time, a signature A has two possible states: it either binds to some object or not.

This is due to the fact that we decided to have *two* language constructs: `bind` $A\ v$ that binds a signature A to an object v, and `unbind` A that unbinds the signature. Our intention was to model these two operations. At the higher-level of abstraction, however, the programmers may want to have a single construct that e.g. replaces software components in one atomic step. Then, there is a guarantee that every signature is always bound to *some* object.

Property 4 (Binding Restriction)**.** At most one object can be bound to a signature A at a time.

If more than one object could be bound to a signature A, then a method call $A.m$ would not know which object to call; similarly, a field access $A.f$ would not know which object to select. (In our language, the same field or method names can appear in different classes.) At the higher-level of abstraction, however, overwriting bindings of A could be encoded; the higher-level `unbind` construct could then remove the current binding and deactivate any previous binding if it exists.

Property 5 (Object Aliasing)**.** An object can be bound to many signatures.

We allow this for expressiveness at the operational semantics. We think that object aliasing could be useful for programmers. If any restriction is required, then it should be declared by programmers, and enforced via a type system.

Property 6 (Failures)**.** If no object is bound to A, then `unbind` A fails, field access $A.f$ fails for any f, and method call $A.m$ fails for any m.

The above property with an exception mechanism built into the calculus allows for more expressiveness. We can express alternative actions in the event of failure at the higher level of abstraction, e.g. "wait till some object is bound".

Property 7 (Concurrency)**.** The operations of binding/unbinding a signature A, and the object field accesses or method calls via A can be concurrent.

Concurrency stems from various reasons, e.g. old and new protocol components may need to coexist for some time, as it was in the case of the switching algorithms in Chapter 5, and the execution of the switching algorithms themselves is concurrent with the execution of the updated protocols.

6.2.2 Semantic properties

Below are two example properties that may be required by some programs with object rebinding.

Property 8 (Reference Consistency)**.** A set of object references $\mathcal{R} = \{A_i.n_j \mid i = 1..k, j = 1..l\}$ is *consistent* in an expression e, if exists object v such that any method call or field access $A_i.n_j$ in $\mathcal{R}$, as part of evaluation of e, refers to v.

This property is useful, e.g. to guarantee that if a method call via a given signature has been executed upon some object, then another reference via this signature (a method call or field access) in the same protocol round will also be executed upon the same object. In Section 6.3, we present an example program that illustrates this case.

Property 9 (Signature Linearity)**.** A signature A is *linear* in a program, if it is either unbound, or it binds the same object v during whole program execution; object v that was bound to A cannot be rebound to other signature.

If a linear signature A has been bound to some object, then it cannot be rebound to another object, and vice versa. This property could be useful in programs in which dynamic object rebinding is not a feature to mask implementations of a given signature, but to authenticate an object via a signature. If objects are communicated between machines (as part of some protocol), it may be useful to use for the authentication an abstract signature of an object, rather than its concrete name.

6.3 Example Program

To explain the need for the reference consistency property (Property 8 in Section 6.2), we use a small, erroneous program. Then, we fix up the program using two synchronization means described in this book, i.e. the `atomic` construct and the concurrency combinators.

The program below implements an exchange of messages between a client and an anonymous server, accessible via a signature `A`. The program instantiates two objects `a` and `b` that implement two different variants of the server, defined by classes `A` and `B`. Initially, `A` binds to `a` but later it is rebound to `b`. Server switching occurs in parallel with the message exchange between the client and the server. For simplicity, the server and the client are not distributed but executed on a single machine.

The client-server protocol uses *public key cryptography*, which can be explained as follows. The client encrypts a message m using server's public key to produce an encrypted message; only the server can decrypt this message, so this ensures secrecy. The server can sign a message m by encrypting it with its secret key (which is the inverse of the public key); any client in possession of server's public key can then decrypt this message. Public key cryptography is used, e.g. in an *authentication protocol* [64]).

```
(* Declaration of signatures and classes *)

sig A
{
  getn : Obj
  put : Int -> Int
}

class P
{
  getn = self       (* an object's unique name *)
  secretKey = 1     (* a secret key of P *)
  Int put (v : Int) =
  {
    decrypt (v, self.secretKey)
  }}

class Q
{
  getn = self       (* an object's unique name *)
  secretKey = 2     (* a secret key of Q *)
  Int put (v : Int) =
  {
    decrypt (v, self.secretKey)
  }}

class Updater
{
  Unit update (x : Sig, o : Obj) =
  {
    unbind x;  (* unbind signature x from any object *)
    bind x o   (* bind signature x to object o *)
  }}

(* Init *)

let a = new P in    (* create object a *)
bind A a;           (* and binds sig A to a *)
let b = new Q in    (* create object b *)

(* The updater *)
fork (new Updater).update(A, b);  (* rebind A to b *)
```

```
(* The client *)
try
  A.put (encrypt (100, keyStore.publicKey(A.getn)))
catch
  0
```

The client (see the clause `try` ... `catch` ...) obtains server's public key from a trusted key store `keyStore`, using a method `keyStore.publicKey`; the method accepts as its argument the server's name, which is obtained by reading a field `A.getn` of the server's object. This name is guaranteed to be unique for every new instantiation of the server (in our program, it is an object location returned by `self`). The key store (omitted here) returns a public key that corresponds to this name. To send a message (a value 100) encrypted using the public key, the client invokes server's method `A.put`. Execution of `A.put` (see class `P` or class `Q`) decrypts the message using server's secret key, which is stored in the server's object field `secretKey`.

An exchange of an encrypted message between the server and the client occurs in parallel with dynamic replacement of the actual object implementing the server. For this, we have an updater object `Updater`, with a single method `update` that implements a simple handover protocol: it takes as arguments a signature and an object, unbinds anything bound to the signature and binds the new object. In the main expression, a concurrent thread (created with `fork`) calls an updater's method `update` that unbinds a server object a (bound to A) and binds server object b to A. For simplicity, we require that A is initially bound. The client does not know if it calls a or b—it is not aware of the hot-swapping done by the updater.

The program is problematic in twofold ways. Firstly, the client may call a server using a signature A that has been unbound by the `update` method and not rebound yet, thus leading to an exception error. Secondly, the following property is not true:

Property 10 (Safety)**.** A message encrypted with a public key of object x is also received by x (for any x).

We would like this property to hold in our program. Otherwise, the client may encrypt and send a message to the server using a public key of another server, which is like an attack on the public key cryptography protocol. Note that if we could make sure that all references to the server via a signature `A` (i.e. methods calls and field accesses via `A`) satisfy the reference consistency property, then safety is preserved. Below are two different implementations of the reference consistency in our program. The first solution uses the `atomic` construct, described in Chapter 3, while the second one uses the concurrency combinators, described in Chapter 4.

6.3.1 Solution using `atomic`

To fix up our program, we encode the message exchange protocol (initiated by the client) and the update protocol (in the `update` method) as two atomic tasks. We assume that the language in this chapter has been extended with the `atomic` and `sync` constructs described in Chapter 3. This extension is straightforward since the core of both languages is the same calculus (the call-by-value λ-calculus). Below is an example code of the corrected program.

```
(* Declaration of verlocks, signatures and classes *)
newlock l : Type in

class Updater
{
  Unit update (x : Sig, o : Obj) =
  {
    atomic l sync l
      (unbind x;    (* unbind signature x from any object *)
       bind x o)    (* and bind signature x to object o
                         atomically *)
  }}

(* Init *)
(...)

(* The updater *)
fork (new Updater).update(A, b);  (* rebind A to b *)

(* The client *)
try
  atomic l sync l A.put(encrypt(100, keyStore.publicKey(A.getn)))
catch
  0
```

The two atomic tasks in the above program use only one verlock of some abstract type `Type`. Thus, this implementation of reference consistency is not different from another implementation that would simply use a single coarse-grain lock instead of atomic tasks. Atomic tasks could turn out to be more useful, e.g., if the client would need to communicate with more than one server using the same secret key.

Consider a variant of our program, in which there are many clients communicating with the server, where each client is executed by a separate thread. The use of the `atomic` construct or locks to implement reference consistency means however that the concurrent clients are also isolated each other. For the program to satisfy safety (see Property 10), it is actually enough to synchronize a client with the updater, but there is no need to synchronize the execution of concurrent clients among themselves. Below is another solution that builds on this observation.

6.3.2 Solution using concurrency combinators

Below we fix up our program using the concurrency combinators. We assume that our language has been extended accordingly, using the constructs and semantics described in Chapter 4, except that the rudimentary binding construct there has been replaced by the `bind` construct introduced in this chapter. Below is an example code of the corrected program.

```
(* Declaration of signatures and classes *)
sig C
{
  update : Sig, Obj -> Unit
}

class Updater
{
  Unit update (x : Sig, o : Obj) =
  {
    unbind x;   (* unbind signature x from any object *)
    bind x o    (* bind signature x to object o *)
  }}

(* Declaration of concurrency combinators *)
D = A.getn ▷ A.put
D isol C.update

(* Init *)
(...)

(* The updater *)
bind C (new Updater);
fork C.update(A, b);   (* rebind A to b *)
```

```
(* The client *)
try
  A.put (encrypt (100, keyStore.publicKey(A.getn)))
catch
  0
```

The core part of the code (implementing the client and the server) is exactly like in the original, erroneous version of our program. The only differences are as follows. We added a definition of a signature for the updater class. Then, we have used the concurrency combinators to declare a synchronization policy, which specifies that a server's method `A.getn` should be called before a method `A.put`, and the execution of these two methods should be atomic with respect to the execution of a method `C.update`. Note that the policy does not require any concurrent callers of methods `A.getn` and `A.put` to be synchronized, i.e. they can be executed in parallel.

We also needed to slightly rewrite the updater's code so that it calls the `update` method via a signature `C` bound to the updater's object; this is required since the declared synchronization policy refers to this object. The code of the client and of the server's classes remains however unchanged.

6.4 Operational Semantics

6.4.1 Basic definitions

We specify the operational semantics of our language using the abstract machine defined in Figures 6.2 and 6.3. The machine evaluates a program by stepping through a sequence of states. A state S consists of three components: an object store Δ, a bind store β, and execution threads T, organized as a sequence $T_0, ..., T_n$.

The *object store* Δ is a finite map from object field selectors to values stored in the fields, where a *field selector*, denoted $o^P.f$, is an object location o^P indexed by a field name f. The *bind store* β is a set of pairs (A, o^P) of a signature name A and an object location o^P bound to the signature. The set difference $\beta \setminus \beta'$ is the set of elements found in β but not found in β'; the union of sets $\beta \cup \beta'$ is the set consisting of the elements of both sets, with no duplicate elements.

We define a small-step evaluation relation $\langle \Delta, \beta \mid e \rangle \longrightarrow \langle \Delta', \beta' \mid e' \rangle$, read "expression e reduces to expression e' in one step, with Δ, β being transformed to Δ', β'". We also use $\longrightarrow^*$ for a sequence of small-step reductions. By *concurrent evaluation*, we mean a sequence of small-step reductions in which the reduction steps can be taken by different threads with possible interleaving.

State Space

$$
\begin{array}{rcllcl}
S & \in & \mathit{State} & = & \mathit{ObjStore} \times \mathit{BindStore} \times \mathit{ThreadSeq} \\
\Delta & \in & \mathit{ObjStore} & = & \mathit{ObjLoc}.\mathit{Sel} \to \mathit{Val} \\
\beta & \in & \mathit{BindStore} & = & \mathit{Sig} \times \mathit{ObjLoc} \\
o^P & \in & \mathit{ObjLoc} & \subset & \mathit{Var} \\
T & \in & \mathit{ThreadSeq} & ::= & e \mid T, T
\end{array}
$$

Evaluation Contexts

$$
\begin{array}{rcl}
\mathcal{E} & = & [\,] \mid \mathcal{E}.n \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \texttt{let}\ x = \mathcal{E}\ \texttt{in}\ e \mid \mathcal{E} := e \mid o^P.f := \mathcal{E} \mid \texttt{bind}\ \mathcal{E}\ e \\
& & \mid \texttt{bind}\ A\ \mathcal{E} \mid \texttt{try}\ \mathcal{E}\ \texttt{catch}\ e \mid \mathcal{E}, T \mid T, \mathcal{E}
\end{array}
$$

Structural Congruence

$$T, T' \equiv T', T$$

$$T, () \equiv T$$

$$\frac{\langle \Delta, \beta \mid T \rangle \longrightarrow \langle \Delta', \beta' \mid T' \rangle}{\langle \Delta, \beta \mid \mathcal{E}[\,T\,] \rangle \longrightarrow \langle \Delta', \beta' \mid \mathcal{E}[\,T'\,] \rangle}$$

$$\frac{T \longrightarrow T'}{\langle \Delta, \beta \mid T \rangle \longrightarrow \langle \Delta, \beta \mid T' \rangle}$$

Transition Relation

$$
\begin{array}{rcl}
\mathit{eval} & \subseteq & ((\mathit{Lab} \to \mathit{Class}) \times \mathit{Exp}) \times \mathit{Val} \\
\mathit{eval}((\textsc{ct}, e), v_0) & \Leftrightarrow & \langle \emptyset, \emptyset \mid e \rangle \longrightarrow^* \langle \Delta, \beta \mid v_0, (), \cdots, () \rangle
\end{array}
$$

Method Body Lookup

$$\frac{\begin{array}{c}\textsc{ct}(P) = \texttt{class}\ P\ \{\overline{f} = \overline{v}, \overline{M}\} \\ t\ m\ F \in \overline{M}\end{array}}{\mathit{mbody}(m, P) = F}$$

Figure 6.2: The DR-calculus: Reduction semantics – Part I

$$\overline{x} : \overline{t} = \{e\}\ \overline{v} \longrightarrow e\{\overline{v}/\overline{x}\} \qquad \text{(R-App)}$$

$$\texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow e\{v/x\} \qquad \text{(R-Let)}$$

$$\frac{\begin{array}{c} o^P \notin dom(\Delta) \\ \textsc{ct}(P) = \texttt{class}\ P\ \{f_1 = v_1, ..., f_k = v_k, \overline{M}\} \\ \Delta' = (\Delta, o^P.f_1 \mapsto v_1, ..., o^P.f_k \mapsto v_k) \end{array}}{\langle \Delta, \beta \mid \texttt{new}\ P \rangle \longrightarrow \langle \Delta', \beta \mid o^P \rangle} \qquad \text{(R-New)}$$

$$\langle \Delta, \beta \mid o^P.f := v \rangle \longrightarrow \langle \Delta[o^P.f \mapsto v], \beta \mid () \rangle \qquad \text{(R-Assign)}$$

$$\langle \Delta, \beta \mid o^P.f \rangle \longrightarrow \langle \Delta, \beta \mid v\{o^P/self\} \rangle \quad \text{if} \ \ \Delta(o^P.f) = v \qquad \text{(R-Field)}$$

$$\frac{mbody(m, P) = F}{o^P.m\ \overline{v} \longrightarrow F\{o^P/self\}\ \overline{v}} \qquad \text{(R-Invk)}$$

$$\texttt{try}\ v\ \texttt{catch}\ e \longrightarrow v \qquad \text{(R-Try)}$$

$$\frac{\texttt{try..catch} \notin \mathcal{E}'}{\texttt{try}\ \mathcal{E}'[\,\texttt{escape}\,]\ \texttt{catch}\ e \longrightarrow e} \qquad \text{(R-Esc)}$$

$$\langle \Delta, \beta \mid \texttt{bind}\ A\ o^P \rangle \longrightarrow \langle \Delta, (\beta \setminus \{(A, \cdot)\}) \cup \{(A, o^P)\} \mid () \rangle \qquad \text{(R-Bind)}$$

$$\langle \Delta, \beta \mid \texttt{unbind}\ A \rangle \longrightarrow \langle \Delta, \beta \setminus \{(A, o^P)\} \mid () \rangle \quad \text{if} \ \ (A, o^P) \in \beta \qquad \text{(R-Unbind-1)}$$

$$\langle \Delta, \beta \mid \texttt{unbind}\ A \rangle \longrightarrow \langle \Delta, \beta \mid \texttt{escape} \rangle \quad \text{if} \ \ (A, \cdot) \notin \beta \qquad \text{(R-Unbind-2)}$$

$$\langle \Delta, \beta \mid A.n \rangle \longrightarrow \langle \Delta, \beta \mid o^P.n \rangle \quad \text{if} \ \ (A, o^P) \in \beta \qquad \text{(R-Lookup-1)}$$

$$\langle \Delta, \beta \mid A.n \rangle \longrightarrow \langle \Delta, \beta \mid \texttt{escape} \rangle \quad \text{if} \ \ (A, \cdot) \notin \beta \qquad \text{(R-Lookup-2)}$$

$$\mathcal{E}[\,\texttt{fork}\ e\,] \longrightarrow \mathcal{E}[\,()\,], e \qquad \text{(R-Fork)}$$

$$v_i, v'_j \longrightarrow v_i \quad \text{if} \ \ i < j \qquad \text{(R-Thread)}$$

Figure 6.3: The DR-calculus: Reduction semantics – Part II

Reductions are defined using evaluation context $\mathcal{E}$ for expressions e. The evaluation context ensures that the left-outermost reduction is the only applicable reduction for each individual thread in the entire program. Context application is denoted by [], as in $\mathcal{E}[\,e\,]$. Structural congruence rules allow us to simplify reduction rules by removing the context whenever possible.

Evaluation of a program $(\textsc{ct}, e)$, where $\textsc{ct}$ is constant, starts in an initial state with empty stores $\emptyset$, and with a single thread that evaluates the expression e. Evaluation then takes place according to the machine's rules in Figure 6.3. The evaluation terminates once all threads have been reduced to values, in which case the value v_0 of the initial, first thread T_0 is returned as the program's result. Subscripts in values reduced from threads denote the sequence number of the thread, i.e. v_i is reduced from i's thread, denoted T_i $(i = 0, 1, ..)$. The execution of threads can be arbitrarily interleaved.

6.4.2 Reduction rules

Below we describe reduction rules in Figure 6.3. The first two evaluation rules are the standard rules of a call-by-value λ-calculus. We write $e\{\overline{v}/\overline{x}\}$ to denote the capture-free substitution of v_i for x_i in the expression e $(i = 1, .., n)$. Function application $\overline{x} : \overline{t} = \{e\}\ \overline{v}$ in (R-App) reduces to the function's body e in which formal arguments $\overline{x}$ are replaced with the actual arguments $\overline{v}$. Execution of $\texttt{let}\ x = v\ \texttt{in}\ e$ in (R-Let) reduces the whole expression to the expression e in which variable x is replaced by value v.

Execution of $\texttt{new}\ P$ creates a new object of class P. The object is identified by a fresh object location o^P, and represented by a new record of object fields $f_1, ..., f_k$ in the object store Δ; see the (R-New) rule. The notation $(\Delta, \overline{o^P.f \mapsto \overline{v}})$ means "the store that maps $\overline{o^P.f}$ to $\overline{v}$ and maps all other selectors to the same thing as Δ". The object fields $f_1, .., f_k$ are accessible via the object location o^P, e.g. $o^P.f_i$ $(i = 1..k)$ refers to a field f_i of object o^P. The object fields in the object record are initialized with field values $v_1, .., v_k$ defined by class P.

Rules (R-Assign) and (R-Field) correspondingly, assign a new value v to the field f of an object o^P, and read the current value stored in an object field $o^P.f$. For instance, let us look at the rule (R-Assign). We use the notation $\Delta[o^P.f \mapsto v]$ to denote update of map Δ at $o^P.f$ to v. Note that the term resulting from this evaluation step is just (); the interesting result is the updated store. The (R-Assign) rule must be applied first, if not possible then we try (R-Field).

Similarly to FJ, the invocation $o^P.m\ \overline{v}$ of a method m of an object o^P applies the beta-reduction rule from the call-by-value λ-calculus; see the (R-Invk) rule. The rule first looks up in the class table $\textsc{ct}$ a method body F of the form $\overline{x} : \overline{t} = \{e\}$ (using a function $mbody(m, P)$ defined in the bottom of Figure 6.2); then, it reduces to the method body in which *self* is replaced by the receiver o^P. Then, the application rule (R-App) (described earlier) can be used, which applies the arguments $\overline{v}$ to the method m.

Exceptions are defined using two rules. The (R-Try) rule defines the case when no exception was thrown; it simply reduces the whole expression `try`...`catch` with the body reduced to a value v to the value v; the `catch` clause is discarded. To throw an exception, the `escape` construct is used. If `escape` is in the redex position of the expression e' in the body of the innermost `try` e' `catch` e, the (R-Esc) rule reduces `try` e' `catch` e to the exception handler e.

Dynamic binder `bind` A o^P in rule (R-Bind) removes from store β any previous binding $(A, \cdot)$ of a signature A, and extends β with a new element of A paired with an object location o^P. The whole expression reduces to the empty value (). Dynamic unbinder `unbind` A in rules (R-Unbind-1) and (R-Unbind-2) removes the binding $(A, \cdot)$ from store β and reduces to the empty value (), or throws an exception with `escape` if no binding of A exists.

Dynamic resolver $A.n$ in rules (R-Lookup-1) and (R-Lookup-2) returns the field/method selector $o^P.n$, where o^P is the object location currently bound to a signature A, or throws an exception if no binding of A exists.

Execution of an expression `fork` e in (R-Fork) creates a new thread which evaluates e; the result of evaluating expression e will be discarded by rule (R-Thread); threads may however have side-effects, e.g. modification of object fields. The results of evaluating threads (except of the initial thread) are discarded by (R-Thread).

6.5 Related Work

6.5.1 Object calculi

There have been many proposals of various object calculi; we sketch some of the most known examples below.

Abadi and Cardelli [2] have developed an imperative calculus of objects, equipped with an operational semantics and typing (and subtyping); with addition of polymorphism, the calculus can express classes and inheritance. The object calculus of Gordon and Hankin [36] extends Abadi and Cardelli's imperative object calculus with operators for concurrency from the π-calculus [74] and operators for synchronization based on mutexes. Our calculus also has synchronization abstractions built-in (the `atomic` construct and concurrency combinators), albeit semantically richer than mutexes.

Igarashi, Pierce and Wadler [53] have proposed a small calculus, Featherweight Java (FJ), that provides classes, methods, fields, inheritance, and dynamic typecasts, with semantics closely following Java's. The design of object features in our calculus has been inspired by FJ, e.g. we have the same rule for method calls, which uses the call-by-value principle of the λ-calculus. However, their calculus omits interfaces and even assignment, while we have assignment and also signatures (which are similar to Java interfaces). On the other hand, we do not model typing and class inheritance since our focus is on the reduction semantics.

The above calculi have been developed mainly to reason about the implementation of objects, object encodings, typing, class inheritance, etc. We are not aware of concurrent object calculi that would have constructs for dynamic object rebinding similar to ours. We discuss some examples of (non-object) calculi with dynamic binding in the next paragraph.

6.5.2 Dynamic rebinding

Dynamic rebinding should not be confused with dynamic linking. *Dynamic linking* of objects in object languages such as Java, refers to resolving object components at runtime (see, e.g. [27] for different models of dynamic linking). Once bound the dynamically linked code usually cannot be rebound, which is different from dynamic object rebinding.

A lot of work on dynamic rebinding appeared in the context of functional languages (see, e.g. work of Moreau [79] on formalization of dynamic binding), focusing either on *dynamic scoping*, in which variable occurrences are resolved with respect to their dynamic environment, or *static scoping with explicit rebinding*, where variables are resolved with respect to their static environment, but additional primitives can be used to explicitly modify these environments.

Dynamic scoping exists in most modern dialects of Lisp, e.g. MIT Scheme's `fluid-let` [77] construct performs dynamically-scoped rebinding of local and global variables; once the construct's expression has been evaluated, the values of the variables are restored. The *quasi-static scoping* Scheme extension of Lee and Friedman [58] has a class of variables, which are initially unresolved. The programmer can use a rebinding primitive to specify new bindings for individual variables. This work is different from ours; we bind whole objects to typed signatures, while the above work is on dynamic binding of variables in functional languages, with a correspondingly different semantics of rebinding.

There are different applications of dynamic rebinding. For instance, dynamic rebinding appears in *marshalling* and *unmarshalling* network messages containing values that refer to local resources. Abstraction-safe marshalling and unmarshalling values between separate programs has been studied by Bierman *et al.* [12], in the context of modules with abstract types; see also the Acute programming language [60]. An extension of Smalltalk with dynamic method redefinition in the scope of *classboxes* is described in [7]; the dynamic rebinding feature is used there to support *software evolution*.

We are not aware of much work on dynamic rebinding in the context of concurrency. The existing implementations are often not satisfactory, e.g. the runtime support of type-safe dynamic Java classes in [66] aborts a thread if a class update is attempted while the thread is executing a method of that class. Our solution is to execute rebindable code fragments and code fragments that do rebinding as concurrent atomic tasks. The semantics of atomicity eliminates the need to abort threads while doing an update.

Chapter 7

Conclusions

In this book we have described language and runtime support, designed for atomicity, synchronization and dynamic protocol update in communicating systems. We have focused on the simplest languages (or calculi) that allow us to study the core problems of Chapter 1; they can provide some basis for any extensions of industrial-strength languages with the support of atomic tasks, declarative synchronization and dynamic object rebinding. All the calculi build on the λ-calculus, which is a theoretical foundation of many present-day programming languages. We think however that analogous work could be carried out for other languages, too. Below we give some concluding remarks and sketch example avenues for future work.

7.1 Atomic Tasks and Versioning

The language of atomic tasks with the runtime system using versioning algorithms described in Chapters 2 and 3 can be useful for encoding atomicity in communicating systems. It allows atomic tasks to perform arbitrary I/O operations, and eliminates the risk of multiple restarts when many atomic tasks compete for the same shared resource (since no task is aborted).

The language of atomic tasks is typed, with a type system able to verify if versioning algorithms have correct input data; this feature provides guarantees that the runtime execution of atomic tasks satisfies isolation. The operational semantics of the language has enabled formal proofs of language safety, including the proof of dynamic correctness of the VA versioning algorithm. For efficiency, the type system could be easily extended to add distinction between read-only and read-write locking. It may be also worthwhile to investigate algorithms for inferring the typing annotations.

For clarity, we have presented somewhat idealised concurrency control algorithms. For instance, if a thread of some task is preempted while holding a verlock then no other thread can access the verlock; this can be solved in any practical implementation using some detection mechanism. In the future, we would like to find more robust ways of implementing `atomic`.

7.2 Declarative Synchronization

The calculus of concurrency combinators in Chapter 4 may be a useful basis for work on different problems of declarative synchronization. One problem that we have identified and solved using a type system, is satisfiability of concurrency combinators. Typing is used to verify if synchronization policy declared using combinators can be satisfied by any execution of the program. This feature could provide some basis for future work on composition safety in modular, communicating systems.

In Chapter 4, we have also demonstrated how to achieve separation of synchronization concerns from functional ones while keeping visible role-oriented aspects of synchronization problems. We described a constraint language for the role-based synchronization. Contrary to similar proposals, the language allows programmers to declare complex synchronization constraints, which may refer to program variables and data structures.

7.3 Dynamic Protocol Update and Rebinding

A model of dynamic protocol update, described in Chapter 5, helped us to clarify basic requirements that are important when implementing support of dynamic protocol update, e.g.: the replaceability property specifies minimal structural (static) requirements on module replacement, and the No-Message-Lost and Message-Order safety properties specify minimal semantic, dynamic requirements during dynamic protocol update.

The model of DPU guided the design of two example DPU algorithms, which are based on synchronized and lazy updating strategies. The algorithms illustrate a trade-off in the design of dynamic protocol update. The former algorithm exhibits strong safety guarantees but requires a subtle distributed infrastructure (totally ordered broadcast) which does not scale to large networks. The latter algorithm scales well but the order of message delivery by updateable service is not respected, which limits its applicability.

A class-based object calculus of dynamic rebinding, described in Chapter 6, could provide basis for the design of component languages for dynamically updateable, communicating systems. We have used a small example program, expressed in the calculus, to illustrate the concept of dynamic object (component) rebinding and other constructs described in this book, such as the `atomic` construct and the concurrency combinators.

By merging the object calculus with the concurrency combinators, we could declare synchronization policy for concurrent objects, with the possibility of replacing the object code at runtime. However, in this case, the static typing rules for verification of combinator satisfiability would have to be redesigned to support dynamic type checking, e.g. *a lá* a mechanism used in Java-like languages for class loading.

Appendix A

Correctness proofs

A.1 Well-typed Programs Satisfy Isolation

A.1.1 Absence of non-declared verlocks

The type system designed for the language of atomic tasks (see Chapter 3) provides rules for proving that in well-typed programs:

(*i*) each task spawned using the `atomic` construct can only read from or write to a reference which is protected by a verlock, and

(*ii*) the verlock itself has been specified in the argument of `atomic`.

Since by the absence of race conditions (Theorem 1 in Section 3.4.1, adopted from the type system for safe locking [31]), in well-typed programs a task cannot access a reference without first obtaining a verlock, we only need to show the second part of the above result. Below we use the semantics to state property (*ii*) formally.

An expression f is *part of* a task $\texttt{task}\ pv\ T$ if $T = \mathcal{E}[\,f\,]$ for some evaluation context $\mathcal{E}$. A task $\texttt{task}\ pv\ T$ *has a version* of a lock l if $pv(l)$ is defined. An expression f *has a version* of a lock l if there exists some task which has a version of l, and f is part of this task. An expression f *requests* a lock location l if $f = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some evaluation context $\mathcal{E}$ and expression e. A task $\texttt{task}\ pv\ T$ is in a *critical section* on a lock location l, if some thread of T is in a critical section on the lock location l.

Now, for the complete language with `atomic` and `task`, the judgment $\vdash_{cs} S$ says in addition to mutual exclusion property stated in Section 3.4.1, that each task being in a critical section on some lock in state S has a version of this lock (see Figure 3.8 and 3.9). According to Lemma 15, the property $\vdash_{cs} S$ is maintained during evaluation.

Lemma 15 (Version-Completeness Preservation)**.** If $\vdash_{cs} S$ and $S \longrightarrow S'$, then $\vdash_{cs} S'$.

Proof. State S may consist of several threads that are evaluated concurrently. Suppose $S = \pi, \sigma \mid \mathcal{E}[\,\texttt{task}\; pv\; T\,]$ for some well-typed store π, σ, context $\mathcal{E}$ and (possibly multithreaded) term T. By rule (R-Task) and evaluation context for `task`, we know that task $\texttt{task}\; pv\; T$ can either reduce to the empty value () if T is a value, or to $\texttt{task}\; pv\; T'$ otherwise, where T' is some expression. The former case is trivial since we have immediately

$$\emptyset \vdash_{cs} () \tag{A.1}$$

by (CS-Empty), which is what we needed.

Let us now consider the latter case. Suppose that task $\texttt{task}\; pv\; T$ is in a critical section on some lock location l. From premise $\vdash_{cs} S$, we have

$$\mathcal{M} \vdash_{cs} \texttt{task}\; pv\; T \tag{A.2}$$

for some $\mathcal{M}$ by (CS-State) and the fact that task $\texttt{task}\; pv\; T$ is a thread in S (by (R-Isol)). But then by (CS-Task)

$$l \in \mathcal{M} \tag{A.3}$$

and version $pv(l)$ is defined. Now we need to consider two subcases, depending on if the reduction step of T enters a new critical section, or not.

Case a). Reduction to a new critical section.
Consider an evaluation step from T to T', such that T has $\texttt{sync}\; l'\; e$ in its redex position. Thus, by rule (R-Sync) $T' = \mathcal{E}'[\,\texttt{insync}\; l'\, e\,]$ and $\pi(l') = 1$ for some context $\mathcal{E}'$, lock location l', and expression e, where $l' \neq l$. Hence, T' is in a critical section on lock l'. Note that by mutual exclusion (Lemma 4) it is not possible to have a reduction step from T to T' if $l' = l$ since (A.2) and (A.3) hold.

Let us assume that $\texttt{task}\; pv\; T'$ does not have a version of lock l', i.e. $pv(l')$ is not defined. But this is not possible, since by version-based protection Lemma 16 (below), if a task $\texttt{task}\; pv\; T$ requests lock location l', then version $pv(l')$ is defined, which contradicts our assumption (since we also know that the private versions map pv is preserved by the reduction step as it is never modified). Thus, $\mathcal{M}' \vdash_{cs} \texttt{task}\; pv\; T'$ and precisely $\mathcal{M}' = \mathcal{M} \uplus \{l'\}$ by

(CS-InSync). From the latter, we have $l \in \mathcal{M}'$ by (A.3).

Case b). No new critical section.
Consider reduction from T to T' such that T has in its redex position an expression other then $\texttt{sync}\ l'\ e$. But then from (A.2) we have $\mathcal{M} \vdash_{cs} \texttt{task}\ pv\ T'$ since T' is in the same critical sections as T, and we know that $l \in \mathcal{M}$ and $pv(l)$ is defined.

From (A.1), a) and b) we obtain the needed result $\vdash_{cs} S'$ by type preservation Corollary 1 (in Section A.1.3) and (CS-State) and induction on threads in S. ☐

Lemma 16 says that a well-typed thread obtains a verlock only when it holds a version of this verlock.

Lemma 16 (Version-Based Protection)**.** Suppose that $\Sigma \mid \Gamma; a; p \vdash f : t$, and f requests a lock location l. Then $\Sigma \mid \Gamma; a; p \vdash l : m$ for some lock type m. Furthermore, there exists a task $\texttt{task}\ pv\ T$ which f is part of, such that $\Sigma \mid \Gamma; a; p \vdash \texttt{task}\ pv\ T : \texttt{Unit}$ and version $pv(l)$ is defined.

Proof. If f requests a lock location l then from the definition of "requesting a lock location" we have $f = \mathcal{E}[\,\texttt{sync}\ l\ e'\,]$ for some evaluation context $\mathcal{E}$ and expression e'. Suppose that $\Sigma \mid \Gamma; a; p \vdash \texttt{sync}\ l\ e' : t'$ for some type t'. Then, by (T-Sync) we have

$$\Sigma \mid \Gamma; a; p \vdash l : m \tag{A.4}$$

for some lock type m, and $m \in a$. From the latter and premise $\Sigma \mid \Gamma; a; p \vdash f : t$, we know that f must be part of some task with allocation a (since $a \neq \emptyset$).

Hence, by (T-Isol) f is reduced from some expression $\texttt{atomic}\ \bar{l}\ e_0$, such that $\Sigma \mid \Gamma; a'; p' \vdash \texttt{atomic}\ \bar{l}\ e_0 : \texttt{Unit}$ (for some a' and p'), where $\bar{l}$ is a sequence of lock locations. Moreover, since allocation a is preserved during task evaluation (since only (T-Isol) can modify a) we have $\Sigma \mid \Gamma; a; \emptyset \vdash e_0 : t''$ for some t'', also by (T-Isol).

From the above, we have immediately $l \in \bar{l}$ by (A.4) and (T-Isol) since $m \in a$. (Note that (T-Isol) is the *only* rule which could add m to allocation a.)

But then, by (R-Isol) expression e_0 can only reduce to $\texttt{task}\ pv\ e_0$ for some pv, such that version $pv(l)$ is defined, which is precisely the needed result since pv is constant and so it does not change while expression e_0 would reduce to T such that $T = \mathcal{E}'[\,f\,]$ for some context $\mathcal{E}'$. By (T-Task), term $\texttt{task}\ pv\ T$ has type $\texttt{Unit}$, which completes the proof. ☐

Note that the above property implies that in our language all lock requests are part of some task. This feature has simplified the type system and reasoning about the isolation property. A full-size language could make a difference between accessing a lock as part of some task, or outside tasks.

We conclude that all verlocks used by each task in well-typed programs are known a priori.

Theorem 10 (Verlock-Usage Predictability)**.** All verlocks that may be requested by a task of a well-typed program are known before the task begins.

Proof. By lock-based protection Lemma 5, it is enough to show that the argument $\overline{l}$ of the `atomic` $\overline{l}\ e$ construct used to spawn a task, is a sequence of all verlocks that *may* be requested by the task. The proof is straightforward by the version-based protection Lemma 16, version-completeness preservation Lemma 15, and induction on tasks and lock location requests. □

The above result implies that the VA algorithm will be able to create upon a task's creation, a private version of each verlock that may be used by the task.

A.1.2 The main result of isolation preservation

We have defined the isolated evaluation for complete tasks (see Section 3.3.3). This is however not a problem since in practice we are interested only in result states of this evaluation. Below we therefore formulate an isolation preservation result for traces that begin and finish in a task-free state. The judgment for such states has the form $\vdash_{tf} S$, read "state S is task-free", which means that either no task has been spawned yet, or if there were any, then they have already completed.

Below we state that each trace of a well-typed program has the "isolation up to" property, provided that the corresponding evaluation finishes in a result state.

Lemma 17 (Isolation Property Up To)**.** Suppose $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash S : t$ and $\vdash_{tf} S$. If $S \longrightarrow^{*} S'$ and $\vdash_{tf} S'$, then the run $S \longrightarrow^{*} S'$ satisfies the isolation property up to S'.

Proof. From premise $\vdash_{tf} S$, we have $\vdash_{cs} S$ by (TF-State). From the latter and premise $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash S : t$, each task in S (if we would let S not to be task-free) is well-typed by (T-State), and by version-based protection Lemma 16, it has versions of all verlocks it may request. Moreover, by version-completeness preservation Lemma 15, we know that this property is preserved by reduction from S to S'' for some state S''. Hence, it is also preserved by any following reductions up to S' (by re-applying Lemma 15). Thus, it holds in all states reached by any tasks that could be spawned by these reductions. But this is precisely one of the two requirements for the correctness of the "isolated evaluation" using the VA algorithm (i.e. Property 12, see A.2.1).

Moreover, from $\vdash_{cs} S$, the lock-based protection Lemma 5 and mutual exclusion Lemma 4 give another requirement (i.e. Property 11, see Section A.2.1) for the correctness of evaluation using the VA algorithm.

By premises $\vdash_{tf} S$ and $\vdash_{tf} S'$, we also know that the evaluation has begun and finished with no active tasks. Hence, by noninterference Theorem 14 (that we prove in Section A.2) and the definition of isolation, we obtain the needed result. □

We conclude that well-typed, terminating programs satisfy the isolation property. A program is *terminating* if all its runs terminate; a run *terminates* if it reduces to a value.

Theorem 11 (Isolation Property)**.** If $\vdash e : t$, then all terminating runs $e \longrightarrow^* v_0$, where v_0 is some value of type t, satisfy the isolation property.

Proof. From premise $\vdash e : t$, e is a closed, well-typed term. Consider any well-typed store π, σ, that is $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash \pi, \sigma$ for some Σ. Then $\vdash \pi, \sigma \mid e : t$ by Definition 10 (see Section A.1.3) and (T-State). Moreover, we have

$$\vdash_{tf} \pi, \sigma \mid e \tag{A.5}$$

since program e (before commencing its execution) does not have any task by syntax (see Figure 3.2). Pick up any terminating trace such that $\pi, \sigma \mid e \longrightarrow^* \pi', \sigma' \mid v_0$ for some store π', σ' and value v_0. From (A.5), we have $\vdash_{cs} \pi', \sigma' \mid v_0$ by (TF-State) and version-completeness preservation (Lemma 15). From the latter, and the fact that $v_0 \neq \texttt{task}\ pv\ T$ for any pv and T, we get $\vdash_{tf} \pi', \sigma' \mid v_0$, which together with (A.5) implies that the run satisfies the isolation property up to v_0 by Lemma 17. Then the result follows by induction on the length of the terminating reduction sequences from $\pi, \sigma \mid e$ to any value. □

We stated our main result for terminating programs. Note however that if a program deadlocks or never terminates, all its runs reaching some result state have the "isolation up to" property (up to this state).

A.1.3 Type soundness

Reduction of a program may either continue forever, or may reach a final state, where no further evaluation is possible. Such a final state represents either an answer or a type error. Since programs expressed in our language are not guaranteed to be deadlock-free, we also admit a deadlocked state to be an (acceptable) answer. Thus, proving type soundness means that well-typed programs yield only well-typed answers.

Our proof of type soundness rests upon the notion of type preservation (also known as subject reduction). The type preservation property states that reductions preserve the type of expressions.

Type preservation by itself is not sufficient for type soundness. In addition, we must prove that programs containing type errors are not typable. We call such expressions with type errors *faulty expressions* and prove that faulty expressions cannot be typed.

Type safety

The statement of the main type preservation lemma must take stores and store typings into account. For this we need to relate stores with assumptions about the types of the values in the stores. Below we define what it means for a store π, σ to be well typed. (For clarity, we omit permissions p from the context and global gv and local lv counters from states when possible.)

Definition 10. A store π, σ is said to be *well typed* with respect to a store typing Σ and a typing context Γ, written $\Sigma \mid \Gamma; a \vdash \pi, \sigma$, if $dom(\pi, \sigma) = dom(\Sigma)$ and $\Sigma \mid \Gamma; a \vdash \mu(l) : \Sigma(l)$ for every store $\mu \in \{\pi, \sigma\}$ and every $l \in dom(\mu)$.

Intuitively, a store π, σ is consistent with a store typing Σ if every value in the store has the type predicted by the store typing.

By canonical forms (Lemma 25 in Section A.1.3), each *location value* $l \in dom(\pi, \sigma)$ can be either a lock location, or a reference location, depending on a concrete type. For simplicity, we often refer to π, σ as the store, meaning individual stores, i.e. either π or σ, depending on a given value and type. If a location value l is a lock location then it is kept in a lock store π; if the value is a reference location then it is kept in a reference store σ.

Type preservation for our language states that the reductions defined in Figures 3.3, 3.4 and 3.5 preserve type:

Theorem 12 (Type Preservation)**.** If $\Sigma \mid \Gamma; a \vdash T : t$ and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ and $\langle \pi, \sigma \mid T\rangle \longrightarrow \langle (\pi, \sigma)' \mid T'\rangle$, then for some $\Sigma' \supseteq \Sigma$, $\Sigma' \mid \Gamma; a \vdash T' : t$ and $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$.

The type preservation theorem asserts that there is some store typing $\Sigma' \supseteq \Sigma$ (i.e., agreeing with Σ on the values of all the old locations) such that a new term T' is well typed with respect to Σ'. This new store typing Σ'

is either Σ or it is exactly $(\Sigma, l : t_0)$, where l is a newly allocated location, i.e. the new element of $dom((\pi, \sigma)')$, and t_0 is the type of the initial value bound to l in the extended store $(\mu, l \mapsto v_0)$ for some $\mu \in \{\pi, \sigma\}$.

Proof. The proof is a straightforward induction on a derivation of $T : t$, using the lemmas below and the inversion property of the typing rules. The proof proceeds by case analysis according to the reduction $T \longrightarrow T'$.

Case $\langle \pi, \sigma \mid \lambda^{b,p} x : s.\, e\ v \rangle \longrightarrow \langle \pi, \sigma \mid e\{v/x\} \rangle$.

From $\Sigma \mid \Gamma; a \vdash \lambda^{b,p} x : s.\, e\ v : t$ we have $\Sigma \mid \Gamma; a \vdash v : s$ and $\Sigma \mid \Gamma; a \vdash \lambda^{b,p} x : s.\, e : s \rightarrow^{b,p} t$ and $b \subseteq a$ by (T-App). From the latter, $\Sigma \mid (\Gamma, x : s); b \vdash e : t$ follows by (T-Fun). Hence $\Sigma \mid \Gamma; b \vdash e\{v/x\} : t$ by substitution Lemma 21 and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ from premise.

Case $\langle \pi, \sigma \mid \texttt{ref}_m\, v \rangle \longrightarrow \langle \pi, (\sigma, r \mapsto v) \mid r \rangle$ if $r \notin dom(\sigma)$.

From $\Sigma \mid \Gamma; a \vdash \texttt{ref}_m\, v : t$ where $t = \texttt{Ref}_m\, t'$, we have

$$\Sigma \mid \Gamma; a \vdash v : t' \tag{A.6}$$

and $\Gamma \vdash m$ by (T-Ref), and $(\Sigma, r : t') \mid \Gamma; a \vdash v : t'$ by store typing Lemma 24, where r is a fresh reference cell location. Hence $(\Sigma, r : t') \mid \Gamma; a \vdash r : \texttt{Ref}_m\, t'$ by (T-RefLoc), which is the first part of the needed result.

From the latter, since $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) and $r : t' \notin \Sigma$ (immediate from the premise that π, σ is well-typed and the assumption that $r \notin dom(\sigma)$) hence $(\Sigma, r : t') \mid \Gamma; a \vdash \pi, (\sigma, r \mapsto v)$ by (A.6) and store extension (Lemma 23), which completes the second part of the needed result.

Case $\langle \pi, \sigma \mid !r \rangle \longrightarrow \langle \pi, \sigma \mid v \rangle$ if $\sigma(r) = v$.

From $\Sigma \mid \Gamma; a \vdash !r : t$, we have $\Sigma \mid \Gamma; a \vdash r : \texttt{Ref}_m\, t$ by (T-Deref). From the latter, we have $\Sigma(r) = t$ and $\Sigma \mid \Gamma \vdash m$ by (T-RefLoc), and so $\Sigma \mid \Gamma; a \vdash \sigma(r) : \Sigma(r)$ by premise that the store π, σ is well typed and Definition 10. Hence $\Sigma \mid \Gamma; a \vdash v : t$ (immediate from the assumption that $\sigma(r) = v$) and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ from premise.

Case $\langle \pi, \sigma \mid r := v \rangle \longrightarrow \langle \pi, \sigma[r \mapsto v] \mid () \rangle$.

From $\Sigma \mid \Gamma; a \vdash r := v : t$ where $t = \texttt{Unit}$, and $\Sigma \mid \Gamma; a \vdash () : \texttt{Unit}$

by (T-Unit), we have immediately the first part of the needed result. From $\Sigma \mid \Gamma; a \vdash r := v : \texttt{Unit}$, we have $\Sigma \mid \Gamma; a \vdash r : \texttt{Ref}_m\, t'$ and

$$\Sigma \mid \Gamma; a \vdash v : t' \tag{A.7}$$

by (T-Assign). From the former, we have $\Sigma(r) = t'$ and $\Sigma \mid \Gamma \vdash m$ by (T-RefLoc), hence $\Sigma \mid \Gamma; a \vdash \pi, \sigma[r \mapsto v]$ by (A.7), premise that the store π, σ is well typed, and the store update Lemma 22, which completes the second part of the needed result.

Case $\langle \pi, \sigma \mid \mathcal{E}[\, \texttt{fork}\; e\,] \rangle \longrightarrow \langle \pi, \sigma \mid \mathcal{E}[\,()\,], e \rangle$.

From $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, \texttt{fork}\; e\,] : t$ we have

$$\begin{aligned} \Sigma' \mid \Gamma'; a &\vdash e : \texttt{Unit} && \text{(A.8)} \\ \Sigma' \mid \Gamma'; a &\vdash \texttt{fork}\; e : \texttt{Unit} && \text{(A.9)} \end{aligned}$$

for some Σ' and Γ' by (T-Fork). From $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, \texttt{fork}\; e\,] : t$ and (A.9) we have $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,()\,] : t$ by (T-Unit) and replacement Lemma 20. From the latter and (A.8) we have $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,()\,], e : t$ by (T-Unit) and (T-Thread), which together with $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), completes both parts of the needed result.

Case $\langle \pi, \sigma \mid f_i, f'_j \rangle \longrightarrow \langle \pi, \sigma \mid f_i \rangle$ if $i < j$.

From $\Sigma \mid \Gamma; a \vdash f_i, f'_j : t$ and $i < j$ we have immediately $\Sigma \mid \Gamma; a \vdash f_i : t$ and $\Sigma \mid \Gamma; a' \vdash f'_j : t'$ for some a' and t' by (T-Thread). The former derivative and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) complete both parts of the needed result.

Case $\langle \pi, \sigma \mid \mathcal{E}[\, \texttt{atomic}\; \bar{l}\; e\,] \rangle \longrightarrow \langle \pi, \sigma \mid \mathcal{E}[\,()\,], \texttt{task}\; pv\; e \rangle$.

From $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, \texttt{atomic}\; \bar{l}\; e\,] : t$, by (T-Isol) we have $\Sigma' \mid \Gamma'; a \vdash l_i : o_{li}$ for all $i = 1..|\bar{l}|$, and $\Sigma' \mid \Gamma'; \{o_{l1}\} \cup ... \cup \{o_{l_{|\bar{l}|}}\} \vdash e : t'$ for some t', and $\Sigma' \mid \Gamma'; a \vdash \texttt{atomic}\; \bar{l}\; e : \texttt{Unit}$ for some Σ' and Γ'. Hence $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\,()\,] : t$ by (T-Unit) and replacement Lemma 20. Since $\Sigma' \mid \Gamma'; a \vdash \texttt{task}\; pv\; e : \texttt{Unit}$ by (T-Task), hence $\langle \pi, \sigma \mid \mathcal{E}[\,()\,], \texttt{task}\; pv\; e \rangle : t$ by (T-Unit) and (T-Thread), which together with $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), completes both parts of the needed result.

Case $\langle \pi, \sigma \mid \texttt{task}\ pv\ v\rangle \longrightarrow \langle \pi, \sigma \mid ()\rangle$.

From $\Sigma \mid \Gamma; a \vdash \texttt{task}\ pv\ v : t$ we have $t = \texttt{Unit}$ by (T-Task), and $\Sigma \mid \Gamma; a \vdash () : \texttt{Unit}$ by (T-Unit), which together with $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), completes both parts of the needed result.

Case $\langle \pi, \sigma \mid \texttt{newlock}\ x : m\ \texttt{in}\ e\rangle \longrightarrow \langle (\pi, l \mapsto 0), \sigma \mid e\{l/x\}\{o_l/m\}\rangle$ if $l \notin dom(\pi)$.

From $\Sigma \mid \Gamma; a \vdash \texttt{newlock}\ x : m\ \texttt{in}\ e : t$ and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), we have $\Sigma \mid (\Gamma, m :: \texttt{Lock}, x : m); a \vdash e : t$ and $\Sigma \mid \Gamma \vdash a$ and $\Sigma \mid \Gamma \vdash t$ by (T-Lock), and hence

$$(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid (\Gamma, m :: \texttt{Lock}, x : m); a \vdash e : t \tag{A.10}$$

by store typing (Lemma 24). Since $(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid \Gamma; a \vdash l : o_l$ by (T-LockLoc), hence $(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid \Gamma; a \vdash e\{l/x\}\{o_l/m\} : t$ by (A.10), substitution (Lemma 21) and the definition of a singleton lock type, which is the first part of the needed result.

From the latter, since $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), $l : \{0,1\} \notin \Sigma$ (immediate from the premise that π, σ is well-typed and the assumption that $l \notin dom(\pi)$), and $\Sigma \mid \Gamma; a \vdash 0 : \{0,1\}$ hence $(\Sigma, l : \{0,1\}, o_l :: \texttt{Lock}) \mid \Gamma; a \vdash (\pi, l \mapsto 0), \sigma$ by store extension (Lemma 23).

Case $\langle \pi, \sigma \mid \texttt{sync}\ l\ e\rangle \longrightarrow \langle \pi[l \mapsto 1], \sigma \mid \texttt{insync}\ l\ e\rangle$ if $\pi(l) = 0$.

From $\Sigma \mid \Gamma; a \vdash \texttt{sync}\ l\ e : t$, we have

$$\Sigma \mid \Gamma; a \ \vdash\ l : o_l \quad o_l \in a \tag{A.11}$$

$$\Sigma \mid \Gamma; a \ \vdash\ e : t \tag{A.12}$$

by (T-Sync). From (A.11) and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), we have

$$\Sigma(l) = \{0, 1\} \tag{A.13}$$

and $\Sigma(o_l) = \texttt{Lock}$ by (T-LockLoc). From (A.11) and (A.12) and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise), we have $\Sigma \mid \Gamma; a \vdash \texttt{insync}\ l\ e : t$ by (T-InSync), which completes the first part of the needed result.

From $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) and (A.13) and $\Sigma \mid \Gamma; a \vdash 1 : \{0,1\}$, we have $\Sigma \mid \Gamma; a \vdash \pi[l \mapsto 1], \sigma$ by the store update Lemma 22, which completes the second part of the needed result.

Case $\langle \pi, \sigma \mid \texttt{insync}\ l\, v\rangle \longrightarrow \langle \pi[l \mapsto 0], \sigma \mid v\rangle$ if $\pi(l) = 1$.

From $\Sigma \mid \Gamma; a \vdash \texttt{insync}\ l\, v : t$, we have

$$\Sigma \mid \Gamma; a \vdash l : o_l \tag{A.14}$$

$$\Sigma \mid \Gamma; a \vdash v : t \tag{A.15}$$

and $o_l \in a$ by (T-InSync), which completes the first part of the needed result. From $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) and (A.14), we have $\Sigma(l) = \{0, 1\}$ and $\Sigma(o_l) = \texttt{Lock}$ by (T-LockLoc). From the latter and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (premise) and $\Sigma \mid \Gamma; a \vdash 0 : \{0, 1\}$, we have $\Sigma \mid \Gamma; a \vdash \pi[l \mapsto 0], \sigma$ by the store update Lemma 22, which completes the second part of the needed result. □

This completes the main part of the proof. It remains to establish several technical lemmas.

Some obvious facts about deductions that we use:

- if $\Sigma \mid \Gamma \vdash \mathcal{E}[\,e\,] : t$ then there exist Σ', Γ' and t' such that $\Sigma' \mid \Gamma' \vdash e : t'$;
- if there are no Σ', Γ' and t' such that $\Sigma' \mid \Gamma' \vdash e : t'$, then there are no Σ, Γ, and t such that $\Sigma \mid \Gamma \vdash \mathcal{E}[\,e\,] : t$.

These follow from the facts that (1) there is exactly one inference rule for each expression form e, and (2) each inference rule requires a proof for each subexpression of the expression in its conclusion.

The first lemma states that we may permute the elements of a context, as convenient, without changing the set of typing elements that can be derived from under it.

Lemma 18 (Permutation)**.** If $\Sigma \mid \Gamma; a \vdash T : t$ and Δ is a permutation of Γ, then $\Sigma \mid \Delta; a \vdash T : t$. Moreover, the latter derivation has the same depth as the former.

Proof. Straightforward induction on typing derivations. □

The following lemma states that extra variables in the typing environment Γ of a judgment $\Gamma \vdash e : t$ that are not free in the expression e may be ignored.

Lemma 19 (Weakening)**.** If $\Gamma(x) = \Gamma'(x)$ for all $x \in fv(e)$ then $\Sigma \mid \Gamma; a \vdash e : t$ iff $\Sigma \mid \Gamma'; a \vdash e : t$.

Proof. Straightforward induction on typing derivations. □

A key lemma that we use in the proof of type preservation is the replacement lemma. It allows the replacement of one of the subexpressions of a typable expression with another subexpression of the same type, without disturbing the type of the overall expression.

Lemma 20 (Replacement)**.** If:

1. $\mathcal{D}$ is a deduction concluding $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, e_1 \,] : t$,
2. $\mathcal{D}'$ is a subdeduction of $\mathcal{D}$ concluding $\Sigma' \mid \Gamma'; a' \vdash e_1 : t'$,
3. $\mathcal{D}'$ occurs in $\mathcal{D}$ in the position corresponding to the hole ($\mathcal{E}$) in $\mathcal{E}[\ ]$, and
4. $\Sigma' \mid \Gamma'; a' \vdash e_2 : t'$

then $\Sigma \mid \Gamma; a \vdash \mathcal{E}[\, e_2 \,] : t$.

Proof. See [122] (for a language with no stores and store typing; the proof is also valid for our language). □

The substitution lemma is the key to showing type preservation for reductions involving substitution.

Lemma 21 (Substitution)**.** If $\Sigma \mid (\Gamma, x : t); a \vdash e : t'$ and $\Sigma \mid \Gamma; a \vdash v : t$, then $\Sigma \mid \Gamma; a \vdash e\{v/x\} : t'$.

Proof. We proceed by induction on a derivation of the statement $(\Gamma, x : t) \vdash e : t'$, and case analysis on the final typing rule used in the proof. (For clarity, we remove store typing Σ, allocation and permission whenever possible.)

Case $e = ()$.

If so then $\Gamma \vdash () : t'$ and $t' = \texttt{Unit}$ by (T-Unit). Then $\Gamma \vdash ()\{v/x\} : t'$ since $()\{v/x\} = ()$ (the same would be for any other constants).

Case $e = x'$.

There are two sub-cases to consider, depending on whether x' is x or another variable.

(1) If $x' \neq x$, then $x' : t' \in \Gamma$ by (T-Var), and $\Gamma \vdash x' : t'$ again by (T-Var). Then $\Gamma \vdash x'\{v/x\} : t'$ since $x'\{v/x\} = x'$.

(2) If $x' = x$, then $x : t' \in \Gamma$ by (T-Var), and $\Gamma \vdash x : t'$ again by (T-Var). Since $x\{v/x\} = v$, hence $\Gamma \vdash x\{v/x\} : t'$.

Case $e = \lambda^{b,p} x' : t_1.\, e_1$.

By (T-Fun), it follows from the assumption $(\Gamma, x : t) \vdash \lambda^{b,p} x' : t_1.\, e_1 : t'$ that $t' = t_1 \rightarrow^{b,p} t_2$ and $(\Gamma, x : t, x' : t_1) \vdash e_1 : t_2$. Using permutation on the given subderivation, we obtain $(\Gamma, x' : t_1, x : t) \vdash e_1 : t_2$. Using weakening (Lemma 19) on the other given derivation $(\Gamma \vdash v : t)$, we obtain $(\Gamma, x' : t_1) \vdash v : t$.

Now, by the inductive hypothesis, $(\Gamma, x' : t_1) \vdash e_1\{v/x\} : t_2$. By (T-Fun), we have $\Gamma \vdash \lambda^{b,p} x' : t_1.\, e_1\{v/x\} : t_1 \rightarrow^{b,p} t_2$. But this is precisely the needed result, since, by the definition of substitution, $\Gamma \vdash (\lambda^{b,p} x' : t_1.\, e_1)\{v/x\} : t_1 \rightarrow^{b,p} t_2$.

Case $e = e_1\ e_2$.

From $(\Gamma, x : t); a \vdash e_1\ e_2 : t'$ by the first premise of (T-App), we have $(\Gamma, x : t); a \vdash e_1 : t_1 \rightarrow^{b,p} t'$ for some t_1 and $b \subseteq a$, and

$$\Gamma; a \vdash e_1\{v/x\} : t_1 \rightarrow^{b,p} t' \tag{A.16}$$

by induction hypothesis. By the second premise of (T-App), we have $(\Gamma, x : t); a \vdash e_2 : t_1$, and

$$\Gamma; a \vdash e_2\{v/x\} : t_1 \tag{A.17}$$

by induction hypothesis.

Then by (T-App) with (A.16) and (A.17) $\Gamma; a \vdash e_1\{v/x\}\ e_2\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (e_1\ e_2)\{v/x\} : t'$.

Case $e = \texttt{ref}_m\, e \colon \texttt{Ref}_m\, t_1$.

By (T-Ref), it follows from the assumption $(\Gamma, x : t) \vdash \texttt{ref}_m\, e : t'$ that $t' = \texttt{Ref}_m\, t_1$, and $(\Gamma, x : t) \vdash e : t_1$ and $\Gamma \vdash m$.

Now, by the induction hypothesis, $\Gamma \vdash e\{v/x\} : t_1$. By (T-Ref), we have $\Gamma \vdash \texttt{ref}_m\, e\{v/x\} \colon \texttt{Ref}_m\, t_1$. But this is precisely the needed result, since, by the definition of substitution $\Gamma \vdash (\texttt{ref}_m\, e)\{v/x\} \colon \texttt{Ref}_m\, t_1$.

Case $e = !e$.

By (T-Deref), it follows from the assumption $(\Gamma, x : t); a \vdash !e : t'$ that $(\Gamma, x : t); a \vdash e : \texttt{Ref}_m\, t'$ for some $m \in a$.

Now, by the induction hypothesis, $\Gamma; a \vdash e\{v/x\} : \mathtt{Ref}_m\, t'$. By (T-Deref), we have $\Gamma; a \vdash !e\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (!e)\{v/x\} : \mathtt{Ref}_m\, t'$.

Case $e = e_1 := e_2$.

From $(\Gamma, x : t); a \vdash e_1 := e_2 : t'$, where $t' = \mathtt{Unit}$, by the first premise of (T-Assign), we have $(\Gamma, x : t); a \vdash e_1 : \mathtt{Ref}_m\, t_1$ for some t_1, and

$$\Gamma; a \vdash e_1\{v/x\} : \mathtt{Ref}_m\, t_1 \tag{A.18}$$

by induction hypothesis. By the second premise of (T-Assign), we have $(\Gamma, x : t); a \vdash e_2 : t_1$ and $m \in a$, and

$$\Gamma; a \vdash e_2\{v/x\} : t_1 \tag{A.19}$$

by induction hypothesis.

Then by (T-Assign) with (A.18) and (A.19) $\Gamma; a \vdash e_1\{v/x\} := e_2\{v/x\} : \mathtt{Unit}$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (e_1 := e_2)\{v/x\} : \mathtt{Unit}$.

Case $e = \mathtt{newlock}\ x' : m\ \mathtt{in}\ e'$.

By (T-Lock), it follows from the assumption $(\Gamma, x : t); a \vdash \mathtt{newlock}\ x' : m\ \mathtt{in}\ e' : t'$ that $(\Gamma, x : t, m :: \mathtt{Lock}, x' : m); a \vdash e' : t'$ and $\Gamma \vdash a$ and $\Gamma \vdash t'$.

Now, by the induction hypothesis, $(\Gamma, m :: \mathtt{Lock}, x' : m); a \vdash e'\{v/x\} : t'$. By (T-Lock), we have $\Gamma; a \vdash \mathtt{newlock}\ x' : m\ \mathtt{in}\ e'\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution, $\Gamma; a \vdash (\mathtt{newlock}\ x' : m\ \mathtt{in}\ e')\{v/x\} : t'$.

Case $e = \mathtt{sync}\ e_1\ e_2$.

From $(\Gamma, x : t); a; p \vdash \mathtt{sync}\ e_1\ e_2 : t'$ by the first premise of (T-Sync), we have $(\Gamma, x : t); a; p \vdash e_1 : m$ and

$$m \in a \ . \tag{A.20}$$

By induction hypothesis

$$\Gamma; a \vdash e_1\{v/x\} : m \ . \tag{A.21}$$

By the second premise of (T-Sync), we have $(\Gamma, x : t); a; p \cup \{m\} \vdash e_2 : t'$. By induction hypothesis

$$\Gamma; a; p \cup \{m\} \vdash e_2\{v/x\} : t' \quad . \tag{A.22}$$

Then by (T-Sync) with (A.20), (A.21) and (A.22) we have $\Gamma; a; p \vdash$ $\texttt{sync}\ e_1\{v/x\}\ e_2\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a; p \vdash (\texttt{sync}\ e_1\ e_2)\{v/x\} : t'$.

Case $e = \texttt{insync}\ e\, f$.

By (T-InSync), it follows from the assumption $(\Gamma, x : t); a; p \vdash \texttt{insync}\ e\, f : t'$ that $(\Gamma, x : t); a; p \vdash e : m$ and $(\Gamma, x : t); a; p \vdash f : t'$ and $m \in a$, $m \in p$.

Now, by induction hypothesis, $\Gamma; a; p \vdash e\{v/x\} : m$ and $\Gamma; a; p \vdash$ $f\{v/x\} : t'$. By (T-InSync), we have $\Gamma; a; p \vdash \texttt{insync}\ e\{v/x\}\, f\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a; p \vdash (\texttt{insync}\ e\, f)\{v/x\} : t'$.

Case $e = \texttt{fork}\ e$.

By (T-Fork), it follows from the assumption $(\Gamma, x : t) \vdash \texttt{fork}\ e : t'$ that $(\Gamma, x : t) \vdash e : t'$ and $t' = \texttt{Unit}$.

Now, by the induction hypothesis, $\Gamma \vdash e\{v/x\} : t'$. By (T-Fork), we have $\Gamma \vdash \texttt{fork}\ e\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma \vdash (\texttt{fork}\ e)\{v/x\} : t'$.

Case $e = \texttt{atomic}\ e_1, ..., e_n\ e_0$.

From $(\Gamma, x : t); a; p \vdash \texttt{atomic}\ e_1, ..., e_n\ e_0 : t'$ and $t' = \texttt{Unit}$, by the first premise of (T-Isol), we have $(\Gamma, x : t); a; p \vdash e_i : m_i$ for all $i = 1..n$, and

$$\Gamma; a; p \vdash e_i\{v/x\} : m_i \quad \text{for all} \ \ i = 1..n \tag{A.23}$$

by induction hypothesis. By the second premise of (T-Isol), we have $(\Gamma, x : t); \{m_1\} \cup ... \cup \{m_n\}; p \vdash e_0 : t_0$ for some t_0, and

$$\Gamma; \{m_1\} \cup ... \cup \{m_n\}; \emptyset \vdash e_0\{v/x\} : t_0 \tag{A.24}$$

by induction hypothesis.

Then by (T-Isol) with (A.23) and (A.24) we have $\Gamma; a; p \vdash$ $\texttt{atomic}\ e_1\{v/x\}, ..., e_n\{v/x\}$

$e_0\{v/x\}$: `Unit`. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a; p \vdash (\texttt{atomic}\ e_1, ..., e_n\ e_0)\{v/x\}$: `Unit`.

Case $e = f_i, f'_j$ and $i < j$.

By (T-Thread), it follows from the assumption $(\Gamma, x : t) \vdash f_i, f'_j : t'$ and $i < j$, that $(\Gamma, x : t) \vdash f_i : t'$ and $(\Gamma, x : t) \vdash f'_j : t''$ for some t''.

Now, by the induction hypothesis, $\Gamma \vdash f_i\{v/x\} : t'$ and $\Gamma \vdash f'_j\{v/x\} : t''$. By (T-Thread) and $i < j$, we have $\Gamma \vdash f_i\{v/x\}, f'_j\{v/x\} : t'$. But this is precisely the needed result, since, by the definition of substitution $\Gamma \vdash (f_i, f'_j)\{v/x\} : t'$.

Case $e = \texttt{task}\ pv\ f$.

By (T-Task), it follows from the assumption $(\Gamma, x : t); a \vdash \texttt{task}\ pv\ f : t'$ where $t' = \texttt{Unit}$, that $a = \{o_{l_1}, ..., o_{l_n}\}$ and $(\Gamma, x : t); a \vdash l_i : o_{l_i}$ and $(\Gamma, x : t); a \vdash pv(l_i) : \textbf{Nat}$ for all $i = 1..n$, and

$$\Gamma; a \vdash pv(l_i)\{v/x\} : \textbf{Nat} \quad \text{for all} \quad i = 1..n \tag{A.25}$$

by induction hypothesis. By the last premise of (T-Task), we have $(\Gamma, x : t); a \vdash f : t$ for some t, and

$$\Gamma; a \vdash f\{v/x\} : t \tag{A.26}$$

by induction hypothesis.

Then by (T-Task) with (A.25, A.26) $\Gamma; a \vdash \texttt{task}\ pv\{v/x\}\ f\{v/x\}$: `Unit`. But this is precisely the needed result, since, by the definition of substitution $\Gamma; a \vdash (\texttt{task}\ pv\ f)\{v/x\}$: `Unit`. $\square$

The next lemma states that replacing the contents of a store with a new value of appropriate type does not change the overall type of the store.

The notation $(\pi, \sigma)[l \mapsto v]$ should be read as $\pi[l \mapsto v], \sigma$ if l is a lock location, or $\pi, \sigma[l \mapsto v]$ if l is a reference cell location. See the canonical forms Lemma 25 in Section A.1.3 that states the possible shapes of values of various types.

Lemma 22 (Store Update)**.** If $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ and $\Sigma(l) = t$ and $\Sigma \mid \Gamma; a \vdash v : t$ then $\Sigma \mid \Gamma; a \vdash (\pi, \sigma)[l \mapsto v]$.

Proof. Immediate from the definition of $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ (see Definition 10). $\square$

The next lemma states that extending the contents of a store with a new value of appropriate type is consistent with the store typing.

The notation $((\pi, \sigma), l \mapsto v)$ should be read as $(\pi, l \mapsto v), \sigma$ if l is a lock location, or $\pi, (\sigma, l \mapsto v)$ if l is a reference cell location. See the canonical forms Lemma 25 in Section A.1.3 that states the possible shapes of values of various types.

Lemma 23 (Store Extension)**.** If $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ and $l : t \notin \Sigma$ and $\Sigma \mid \Gamma; a \vdash v : t$ then $(\Sigma, l : t) \mid \Gamma; a \vdash ((\pi, \sigma), l \mapsto v)$.

Proof. Immediate from the definition of $\Sigma \mid \Gamma; a \vdash \pi, \sigma$. □

Finally, we need a kind of weakening lemma for stores, stating that, if a store is extended with a new location then the extended store still allows us to assign types to all the same terms as the original.

Lemma 24 (Store Typing)**.**
If $\Sigma \mid \Gamma; a \vdash e : t$ and $\Sigma' \supseteq \Sigma$, then $\Sigma' \mid \Gamma; a \vdash e : t$.

Proof. Easy by induction. □

A corollary of Type Preservation (Theorem 12) is that reduction steps preserve type.

Corollary 1 (Type Preservation)**.** If $\Sigma \mid \Gamma; a \vdash T : t$ and $\Sigma \mid \Gamma; a \vdash \pi, \sigma$ and $\langle \pi, \sigma \mid T \rangle \longrightarrow^* \langle (\pi, \sigma)' \mid T' \rangle$, then for some $\Sigma' \supseteq \Sigma$, $\Sigma' \mid \Gamma; a \vdash T' : t$ and $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$.

Proof. If $\langle \pi, \sigma \mid T \rangle \longrightarrow \langle (\pi, \sigma)' \mid T' \rangle$, then $T = \mathcal{E}[\, e_1 \,]$ and $T' = \mathcal{E}[\, e_2 \,]$, and $\langle \pi, \sigma \mid e_1 \rangle \longrightarrow \langle (\pi, \sigma)' \mid e_2 \rangle$ and $\Sigma' \mid \Gamma; a \vdash (\pi, \sigma)'$, for some $\Sigma' \supseteq \Sigma$, so $\Sigma' \mid \Gamma; a \vdash T' : t$ by the replacement Lemma 20. Then the result follows by induction on the length of the reduction sequence $\langle \pi, \sigma \mid T \rangle \longrightarrow^* \langle (\pi, \sigma)' \mid T' \rangle$. □

Evaluation progress

Subject reduction ensures that if we start with a typable expression, then we cannot reach an untypable expression through any sequence of reductions. This by itself, however, does not yield type soundness.

Below, we prove that evaluation of a typable expression cannot get *stuck*, i.e. either the expression is a value or there is some reduction defined. However, we do allow reduction to be suspended indefinitely since our language is not deadlock-free. This is acceptable since we define and guarantee isolation, respectively isolation-up-to, only for programs that either terminate, or reach some result state (see Theorem 11 and Lemma 17).

A canonical forms lemma states the possible shapes of values of various types.

Lemma 25 (Canonical Forms)**.**
1) If v is a value of type $\texttt{Unit}$, then v is $()$.
2) If v is a value of type $t \to^{a,p} s$, then $v = \lambda^{a,p} x : t.\, e$.
3) If v is a value of type m, then v is a lock location.
5) If v is a value of type $\texttt{Ref}_m\, t$, then v is a reference cell location (or reference location, in short) of a reference cell storing values of type t.

Proof. Straightforward from the grammar in Figure 3.2 and the extended grammar in Figure 3.3. □

We state progress only for closed expressions, i.e. with no free variables. For open terms, the progress theorem fails. This is however not a problem since complete programs—which are the expressions we actually care about evaluating—are always closed.

Independently of the type system and store typing, we should define which state we regard as well-formed. Intuitively, a state is well-formed if the content of the store is consistent with the expression executed by the thread sequence. (We omit global and local counters that are also part of the state, as they are not represented in expressions explicitly.) In case of store π, if there is some evaluation context $\mathcal{E}[\,\texttt{insync}\ l\, e\,]$ in the thread sequence for any lock location l, then $\pi(l)$ should contain 1, marking that the lock has been acquired. As for the store σ, containing the content of each reference cell, we may only require that it is well typed.

Definition 11. Suppose π, σ is a well-typed store, and $\overline{f}$ is a well-typed sequence of expressions, where each expression is evaluated by a thread. Then, a state $\pi, \sigma \mid \overline{f}$ is *well-formed*, denoted $\vdash_{wf} \pi, \sigma \mid \overline{f}$, if for each expression f_i ($i < |\overline{f}|$) such that $f_i = \mathcal{E}[\,\texttt{insync}\ l\, e\,]$ for some l, there is $\pi(l) = 1$.

Of course, a well-typed, closed expression with empty store is well-formed.

According to Lemma 26, the property $\vdash_{wf} \pi, \sigma \mid \overline{f}$ is maintained during evaluation.

Lemma 26 (Well-Formedness Preservation)**.** If $\vdash_{wf} \pi, \sigma \mid \overline{f}$ and $\pi, \sigma \mid \overline{f} \longrightarrow (\pi, \sigma)' \mid \overline{f}'$ then $\vdash_{wf} (\pi, \sigma)' \mid \overline{f}'$.

Proof. Consider a well-formed state $\pi, \sigma \mid e_0$, for some well-typed program $\vdash e_0 : t$ and well-typed store π, σ. Suppose that $e_0 = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some context $\mathcal{E}$, and $\pi(l) = 0$. (Note that when a lock location l is created, then initially $\pi(l) = 0$ by (R-Lock).) From the latter and premise that the state is well-formed, we know that there is no context $\mathcal{E}'$ such that $e_0 = \mathcal{E}'[\,\texttt{insync}\ l\, e'\,]$ for any e'. From the latter and premise, by (R-Sync), we could reduce expression e_0 to $(\pi, \sigma)' \mid e_1$, such that $e_1 = \mathcal{E}[\,\texttt{insync}\ l\, e\,]$. But then, after reduction step, we have $\pi(l) = 1$ (again by (R-Sync)). Moreover, by type preservation Theorem 12, the new state is well typed. Thus, from the definition of well-formedness, we get immediately that $\vdash_{wf} (\pi, \sigma)' \mid e_1$. Finally, we obtain the needed result by induction on thread creation. □

A state $\pi, \sigma \mid T$ is *deadlocked* if there exist only evaluation contexts $\mathcal{E}$, such that $T = \mathcal{E}[\,\texttt{sync}\ l\ e\,]$ for some verlocks l, such that $\pi(l) = 1$ for each l (i.e. the verlocks are not free) and there is no other evaluation context possible.

Now, we can state the progress theorem.

Theorem 13 (Progress)**.** Suppose T is a closed, well-typed term (that is, $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash T : t$ for some t and Σ). Then either T is a value or else, for any store π, σ such that $\Sigma \mid \emptyset; \emptyset; \emptyset \vdash \pi, \sigma$ and $\vdash_{wf} \pi, \sigma \mid T$, there is some term T' and store $(\pi, \sigma)'$ with $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, or else T is deadlocked on some lock(s).

Proof. Straightforward induction on typing derivations. We need only show that either $\pi, \sigma \mid T \longrightarrow (\pi, \sigma)' \mid T'$, or T is a value, or $\pi, \sigma \mid T$ is a deadlocked state. From the definition of $\longrightarrow$, we have $T \longrightarrow T'$ iif $T = \mathcal{E}[\,e_1\,]$, $T' = \mathcal{E}[\,e_1'\,]$, and $e_1 \longrightarrow e_1'$.

Case The variable case cannot occur (because e is closed).
Case The abstract case is immediate, since abstractions are values.
Case $T = e_1\ e_2$ with $\vdash e_1 : t \rightarrow^{b,p} s$ and $\vdash e_2 : t$
By the induction hypothesis, either e_1 is a value or else it can make a step of evaluation, and likewise e_2, or T is a deadlocked state. If e_1 can take a step, then $e_1 = \mathcal{E}_1[\,e'\,]$ and $e' \longrightarrow e''$. But then $T = \mathcal{E}[\,e'\,]$ where $\mathcal{E} = \mathcal{E}_1\ e_2$; thus $T \longrightarrow \mathcal{E}[\,e''\,]$. Otherwise, e_1 is a value. If e_1 is a value and e_2 can take a step, then $e_2 = \mathcal{E}_2[\,e'\,]$ and $e' \longrightarrow e''$ then $T = \mathcal{E}[\,e'\,]$ where $\mathcal{E} = e_1\ \mathcal{E}_2$; thus $T \longrightarrow \mathcal{E}[\,e''\,]$. Otherwise, e_1 and e_2 are values, or T is a deadlocked state. Finally, if both e_1 and e_2 are values, then the canonical forms lemma tells us that e_1 has the form $\lambda^{b,p} x : t.\, e_1'$, and so rule (R-App) applies to T.

Other cases are straightforward induction on typing derivations, following the pattern of the case with $T = e_1\ e_2$. □

A.2 Dynamic Correctness of the VA Algorithm

Independently of the type system, we must prove that our example versioning algorithm (VA), which is used for scheduling of task operations, is correct, i.e. it can be actually used to evaluate programs so that all possible executions satisfy the isolation property.

A.2.1 Assumptions and definitions

The VA algorithm is correct only for programs that have the following two properties:

Property 11. All data accesses are protected by verlocks.

Property 12. Each task has a version of each verlock it may use.

But these two properties correspond precisely to the absence of race freedom, and the absence of undeclared verlocks properties. We have shown that they hold for all well-typed programs (see Theorems 1 and 10). Thus, to prove the correctness of the VA algorithm, it remains to show that all tasks of a well-typed program never interfere (from the definition of isolation).

From the definition of `sync` l e, we know that a locked expression e can be executed only by a single thread since other threads would be blocked (due to the atomicity property of locks). Moreover, by the absence of race conditions Theorem 1, we know that in order to access a reference, first a verlock must be taken. Therefore, we can formulate the definition of noninterference using verlocks instead of references:

Definition 12 (Noninterference)**.** Tasks in a concurrent run *do not interfere* (or satisfy the *noninterference* property) if there exists some ideal serial run R^{s} of all these tasks, such that given any verlock, the order of acquiring the verlock by tasks in the concurrent run is the same as in R^{s}.

Below we explain each step of the algorithm, given by evaluation rules *VA0-3* in Figure 3.5. We require steps *VA1* and *VA2* to be atomic. We write gv_l and lv_l as shorthand for $gv(l)$ and $lv(l)$.

*VA0***:** Upon lock creation, by rule (R-Lock), initialize global and local counters of the new lock to zero.

*VA1***:** At the moment of spawning a new task k using `atomic` $\bar{l}$ e, by rule (R-Isol), for each lock l_i where $i = 1, .., |\bar{l}|$, that may be requested by this task, increase counter gv_{l_i} by one. Create a fresh (read-only) map pv_k that contains bindings from the locks l_i to their upgraded versions gv_{l_i}.

VA2: A task k can acquire a lock l only when, by rule (R-Sync), the lock is free and the task holds a (private) version of this lock that—when downgraded by one—matches the current (local) version maintained by the lock, i.e.

$$pv_k(l) - 1 = lv_l \quad . \tag{A.27}$$

VA3: After a task k has completed its execution, i.e. all threads of the task have terminated, by rule (R-Task), for each lock l_i, where $i = 1, .., |\bar{l}|$, wait until condition (A.27) is true, then upgrade a local version of each lock l_i, so that $lv_{l_i} = pv_k(l_i)$.

Essentially, the VA algorithm implements ordering of lock acquisitions based on versions. Tasks acquire verlocks in such order as is required to satisfy the noninterference property. We need to show that all possible evaluations of a typable expression cannot lead to a task-free state that is not obtainable by some serialized evaluation of tasks. Note that we do not require a program to terminate. However, we consider its correctness only for a set of tasks that will eventually terminate.

To prove the correctness of the algorithm, we only need to show that all tasks of each well-typed program never interfere (from the definition of isolation).

The proof proceeds by proving lemmas about safety and liveness properties of verlocks, verlock-based mutual exclusion, and finally about ordering properties of verlock-based access to references. We begin from introducing a few definitions.

For a task `task` $pv\ e$ where $pv(l)$ is defined, we define *access* of this task to a verlock l, denoted a, as a pair $(pv(l), lv_l)$, where $pv(l)$ and lv_l are correspondingly, a private and local versions of verlock l. Access of `task` $pv\ e$ to a lock l is *defined* if $pv(l)$ is defined.

Access $a_k = (pv_k(l), lv_l)$ of a task k is *valid* if condition (A.27) is true. A task *gets* a valid access $(pv_k(l), lv_l)$ when condition (A.27) is becoming true.

A.2.2 Verlock access

Lemma 27 (Verlock Safety)**.** A verlock can be acquired only by a task which has valid access to the verlock.

Proof. Straightforward from the definition of access and the premise of (R-Sync). □

Lemma 28 (Access Liveness)**.** Each access of a given task in a concurrent run will be eventually valid, provided that all tasks terminate.

Proof. Let k_0 be the first task, with access a_{k_0} to some verlock l defined. By steps *VA0* and *VA1*, $a_{k_0} = (pv_{k_0}(l), lv_l)$, where $pv_{k_0}(l) = 1$ and $lv_l = 0$. Moreover, access a_{k_0} is valid since condition (A.27) is true. Consider a task k_1 created after k_0, with access a_{k_1} to l defined, where $a_{k_1} = (2, 0)$. The access a_{k_1} is not valid since (A.27) is false $(2 - 1 \neq 0)$. However, since we assumed that tasks terminate, then by step *VA3*, the local version of verlock l will be eventually upgraded by 1 as soon as k_0 terminate. But then a_{k_1} is valid. Hence, by induction on tasks, we will get the needed result. ☐

Lemma 29 (Verlock Liveness)**.** Each non-free verlock requested by a task will be eventually acquired, provided that it will be released.

Proof. Straightforward from access liveness Lemma 28 and the premise of (R-Sync). ☐

Lemma 30 (Private-Version Uniqueness)**.** Each task has a unique private version of each verlock during task lifetime.

Proof. Immediate from step *VA1*, where for each verlock l, $pv(l)$ is given a value equal gv_l increased by one, and the fact that step *VA1* is atomic and $pv(l)$ is constant. ☐

Lemma 31 (Access Uniqueness)**.** For each verlock and any task which has access to this verlock defined, the access is globally unique.

Proof. Immediate from the definition of access and the private version uniqueness Lemma 30. ☐

Lemma 32 (Valid-Access Mutual Exclusion)**.** At any time, there is only one access to a given verlock which is valid.

Proof. Consider a verlock l. Since local version lv_l of this verlock is the same for all tasks at any time, from private-version uniqueness Lemma 30, we have that at any given time, there is only one task which can have access for which validity condition (A.27) is true. Hence, we obtain the needed result. ☐

Lemma 33 (Access Privacy)**.** A valid access a_k of a task k can be invalidated only by task k.

Proof. Consider a valid access $a_k = (pv_k(l), lv_l)$ of some task k to a verlock l. By access uniqueness Lemma 31, there is no other task k' with access $(pv_{k'}(l), lv_l)$ such that $pv_{k'}(l) = pv_k(l)$. On the other hand, from valid-access mutual exclusion Lemma 32, we know that it is not possible that some other task could have (different) access to verlock l that is also valid. Thus, we know that only k has a valid access to l. Moreover, by step *VA3* we know that task k can only upgrade lv_l if (A.27) is true. It means that lv_l can only be upgraded if k has a valid access a_k to l. But this is precisely the needed result, since by modifying lv_l access a_k to l is no longer valid. □

Lemma 34 (Valid-Access Preservation)**.** If a task has got valid access to a verlock, then it will have valid access to it at any time (until it would invalidate it).

Proof. Straightforward from valid-access mutual exclusion Lemma 32 and access privacy Lemma 33. □

Lemma 35 (Verlock-Set Mutual Exclusion)**.** As long as a task is allowed to acquire a verlock l, no other task can acquire verlock l.

Proof. Straightforward from valid-access-preservation Lemma 34 and verlock safety Lemma 27. □

By verlock-set mutual exclusion Lemma 35, and the fact that we are not interested in the relative order of lock acquisitions made by the same task (since *any* such order would satisfy Definition 12 of noninterference), we can represent all acquisitions of a given verlock made by a given task by any single such acquisition. Thus, in the rest of the proof, we can consider a system in which each verlock is acquired by a task at most once. By Lemma 35, the proven result will be valid for any system.

A.2.3 Access ordering

Lemma 36 (Access Ordering)**.** The order of acquiring a verlock by tasks corresponds to the order in which tasks got valid access to it.

Proof. Immediate by verlock safety Lemma 27 and verlock-set mutual exclusion Lemma 35. □

Lemma 37 (Valid-Access Ordering)**.** The relative order of getting valid access to a verlock by tasks corresponds to the order of creating the tasks.

Proof. Consider a task k, which gets valid access to some lock l. Access becomes valid when condition (A.27) becomes true. By step *VA3*, this occurs when some other task k' upgrades a local version lv_l by 1. By access privacy and valid-access mutual exclusion, the task k' has valid access to l and is the only one which has it. The valid access of k' becomes invalidated after upgrading lv_l by 1, and then given to k. From the latter and (A.27), we can derive that

$$pv_{k'}(l) = pv_k(l) - 1 \ . \tag{A.28}$$

Moreover, from step *VA1*, we know that the order of private versions corresponds to the order of creating tasks, i.e. if k_i has been created before k_j, then $pv_{k_i}(l) < pv_{k_j}(l)$ for each lock l such that both tasks have defined access to it. Hence, from (A.28), we know that k' has been created before k. Finally, by induction on tasks we obtain the needed result. □

Lemma 38 (Total Ordering)**.** The relative order of acquiring a verlock by tasks is the same for every verlock.

Proof. Immediate from verlock safety Lemma 27, verlock-set mutual exclusion Lemma 35, and access ordering Lemma 36, valid-access ordering Lemma 37, and the fact that the order of creating tasks is total (by step *VA1*). □

Lemma 39 (Natural Ordering)**.** The order of acquiring verlocks by tasks in a concurrent run is the same as in some serial run.

Proof. By the definition of a serial run of tasks, we have immediately that all verlocks are acquired by the tasks in the order in which the tasks have been created (let's call this property a "natural order").

From verlock safety Lemma 27, valid-access ordering Lemma 37, verlock-set mutual exclusion Lemma 35, and total ordering Lemma 38, it is straightforward that any concurrent run has the "natural order" property. Moreover, since we only consider isolation for expressions that reached a task-free state (see Lemma 17), hence we are allowed to consider only concurrent runs in which all tasks terminate. This means that each verlock acquired must be eventually released (note that all verlocks are initially free by (R-Lock)). Thus, by verlock liveness Lemma 29, all verlocks requested will be eventually acquired. From the latter, we conclude that there can be a plausible serial run considered, and obtain the needed result. □

A.2.4 Isolated task execution

We conclude that the VA algorithm can be used to implement the isolated execution of tasks.

Theorem 14 (Noninterference)**.** If a program has Properties 11 and 12, then any evaluation of the program up to any result state, using the VA algorithm, satisfies the noninterference property.

Proof. By natural ordering Lemma 39, the noninterference property is satisfied in any concurrent run in which verlocks are acquired when permitted by the algorithm, which completes the proof. □

Bibliography

[1] *Appia*. `http://appia.di.fc.ul.pt/`.

[2] M. Abadi and L. Cardelli. An imperative object calculus. In *Proceedings of TAPSOFT '95: Theory and Practice of Software Development, the 6th International Joint Conference CAAP/FASE*, volume 915 of *LNCS*. Springer, May 1995.

[3] F. Achermann and O. Nierstrasz. Applications = Components + Scripts – A Tour of Piccola. In M. Aksit, editor, *Software Architectures and Component Technology*, pages 261–292. Kluwer, 2001.

[4] J. Armstrong and R. Virding. Erlang – an experimental telephony switching language. In *Proceedings of XIII International Switching Symposium*, May–June 1991.

[5] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition*. Addison Wesley, 2000.

[6] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In *Proceedings of ECOOP '02: the 16th European Conference on Object-Oriented Programming*, volume 2374 of *LNCS*. Springer, June 2002.

[7] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of JMLC '03: the Joint Modular Languages Conference*, volume 2789 of *LNCS*. Springer, Aug. 2003.

[8] M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. *Electronic Notes in Theoretical Computer Science*, 39(1), 2000.

[9] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] M. Bickford, C. Kreitz, R. van Renesse, and R. L. Constable. An experiment in formal design using meta-properties. In *Proceedings of DISCEX-II '01: the 2nd DARPA Information Survivability Conference and Exposition*, June 2001.

[11] G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proceedings of USE '03: the Second International Workshop on Unanticipated Software Evolution*, Apr. 2003.

[12] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In *Proceedings of ICFP '03: the ACM SIGPLAN International Conference on Functional Programming*, Aug. 2003.

[13] A. P. Black, V. Cremet, R. Guerraoui, and M. Odersky. An equational theory for transactions. In *Proceedings of FSTTCS '03: the 23rd International Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 2914 of *LNCS*. Springer, Dec. 2003.

[14] G. S. Blair, L. Blair, V. Issarny, P. Tuma, and A. Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Proceedings of Middleware 2000: the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, volume 1795 of *LNCS*. Springer, Apr. 2000.

[15] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: theory and practice. *Software Engineering Journal*, 8(2):102–108, 1993.

[16] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proceedings of POPL '05: the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2005.

[17] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[18] *Cactus*. `http://www.cs.arizona.edu/cactus/`.

[19] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[20] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[21] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of ICDCS '01: the 21st IEEE International Conference on Distributed Computing Systems*, Apr. 2001.

[22] P. K. Chrysanthis and K. Ramamritham. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1990.

[23] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with 'readers' and 'writers'. *Communications of the ACM*, 14(10):667–668, Oct. 1971.

[24] X. Défago, A. Schiper, and P. Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Technical Report DSC-2000-036, Communication Systems Department, EPFL, Sept. 2000.

[25] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[26] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, 1968.

[27] S. Drossopoulou, G. Lagorio, and S. Eisenbach. Flexible models for dynamic linking. In *Proceedings of ESOP '03: the European Symposium on Programming*, Apr. 2003.

[28] D. Duggan. Type-based hot swapping of running modules. In *Proceedings of ICFP '01: the 6th ACM SIGPLAN International Conference on Functional Programming*, Sept. 2001.

[29] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.

[30] F. L. Fessant and L. Maranget. Compiling join-patterns. In *Proceedings of HLCL '98: the 3rd International Workshop on High-Level Concurrent Languages*, 1998.

[31] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of ESOP '99: the 8th European Symposium on Programming*, volume 1576 of *LNCS*. Springer, Mar. 1999.

[32] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of PLDI '03: Conference on Programming Language Design and Implementation*, June 2003.

[33] *Fortika*. `http://lsrwww.epfl.ch/crystall/`.

[34] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96: the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1996.

[35] S. Frølund and G. Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP '93: the 7th European Conference on Object-Oriented Programming*, volume 627 of *LNCS*, July 1993.

[36] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings of HLCL'98: the 3rd International Workshop on High-Level Concurrent Languages*, Elsevier ENTCS 16(3), 1998.

[37] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[38] M. H. Graham. Issues in real-time data management. Technical Report SEI-TR-17, Software Engineering Institute, Carnegie-Mellon University, July 1991.

[39] C. A. Gunter. *Semantics of Programming Languages – Structures and Techniques*. MIT Press, 1992.

[40] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.

[41] N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1719–1736, Nov. 1994.

[42] J. Hallstrom, W. Leal, and A. Arora. Scalable evolution of highly available systems. *Transactions of the IEICE: the Institute for Electronics, Information and Communication Engineers, IEICE/IEEE Joint Special Issue on Assurance Systems and Networks*, E86-B(10):2154–2166, 2003.

[43] R. Harper. Practical foundations for programming languages. In preparation. Draft available on-line, 2007.

[44] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA '03: the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[45] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of PPoPP '05: the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[46] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[47] M. W. Hicks, J. T. Moore, and S. Nettles. Dynamic software updating. In *Proceedings of PLDI '01: the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[48] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71. Academic Press, 1972.

[49] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, 1972.

[50] *Java HotSpot*. `http://java.sun.com/products/hotspot/`.

[51] W. Hürsch and C. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Feb. 1995.

[52] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[53] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of OOPSLA '99: the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications*, Nov. 1999.

[54] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proceedings of ECOOP '03: the 17th European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*. Springer, July 2003.

[55] S. Jagannathan and J. Vitek. Optimistic concurrency semantics for transactions in coordination languages. In *Proceedings of COORDINATION '04: the 6th International Conference on Coordination Models and Languages*, volume 2949 of *LNCS*. Springer, Feb. 2004.

[56] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, Oct. 2001.

[57] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of Middleware 2000: the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, volume 1795 of *LNCS*, April 2000.

[58] S.-D. Lee and D. P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proceedings of POPL '93: the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1993.

[59] Y.-F. Lee and R.-C. Chang. Developing dynamic-reconfigurable communication protocol stacks using Java. *Software Practice & Experience*, 35(6):601–620, 2005.

[60] J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of ICFP '03: the ACM SIGPLAN International Conference on Functional Programming*, 2003.

[61] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98: Conference on Object-oriented programming, systems, languages, and applications*, Oct. 1998.

[62] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting meta-properties. In *Proceedings of WARGC '01: Workshop on Applied Reliable Group Communication*, Apr. 2001.

[63] C. V. Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, Dec. 1997 (1998).

[64] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS '96: Workshop on Tools and*

Algorithms for the Construction and Analysis of Systems, volume 1055 of *LNCS*, Mar. 1996.

[65] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[66] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of ECOOP 2000: The European Conference on Object-Oriented Programming*, volume 1850 of *LNCS*. Springer, June 2000.

[67] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[68] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[69] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. Cactus: Comparing protocol composition frameworks. In *Proceedings of SRDS '03: the 22nd Symposium on Reliable Distributed Systems*, Oct. 2003.

[70] S. Mena, A. Schiper, and P. T. Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of Middleware '03: the 4th ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 414–432. Springer, June 2003.

[71] S. Mena de la Cruz. *Protocol Composition Frameworks and Modular Group Communication: Models, Algorithms and Architectures*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (EPFL), Nov. 2006.

[72] G. Milicia and V. Sassone. Jeeg: A programming language for concurrent objects synchronization. In *Proceedings of ACM Java Grande/ISCOPE Conference*, Nov. 2002.

[73] G. Milicia and V. Sassone. Jeeg: Temporal constraints for the synchronization of concurrent objects. Tech. Report RS-03-6, BRICS, Feb. 2003.

[74] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[75] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML – Revised*. MIT Press, 1997.

[76] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of ICDCS '01: the 21st IEEE International Conference on Distributed Computing Systems*, Apr. 2001.

[77] MIT. *Scheme*. `http://www.swiss.ai.mit.edu/projects/scheme/`.

[78] J. Mocito, L. Rosa, N. Almeida, H. Miranda, L. Rodrigues, and A. Lopes. Context adaptation of the communication stack. In *Proceedings of MDC '05: the 3rd International Workshop on Mobile Distributed Computing*, June 2005.

[79] L. Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, Dec. 1998.

[80] *Objective Caml*. `http://caml.inria.fr`.

[81] P. Panangaden and J. Reppy. The Essence of Concurrent ML. In F. Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation, and Application*, Monographs in Computer Science, pages 5–29. Springer, 1997.

[82] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[83] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of POPL '96: the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1996.

[84] B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[85] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, Cambridge, MA, 2005.

[86] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[87] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[88] R. Ramirez and A. E. Santosa. Declarative concurrency in Java. In *Proceedings of HIPS 2000: the 5th Workshop on High-Level Parallel Programming Models and Supportive Environments*, May 2000.

[89] R. Ramirez, A. E. Santosa, and R. H. C. Yap. Concurrent programming made easy. In *Proceedings of ICECCS 2000: the 6th IEEE International Conference on Engineering of Complex Computer Systems*, Sept. 2000.

[90] S. Ren and G. A. Agha. RTsynchronizer: Language support for real-time specifications in distributed systems. In *Proceedings of ACM Workshop on Languages, Compilers, & Tools for Real-Time Systems*, June 1995.

[91] O. Rütti, P. T. Wojciechowski, and A. Schiper. Dynamic update of distributed agreement protocols. Technical Report IC-2005-012, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Mar. 2005.

[92] O. Rütti, P. T. Wojciechowski, and A. Schiper. Service Interface: A new abstraction for implementing and composing protocols. In *Proceedings of SAC '06: the 21st ACM Symposium on Applied Computing – Track on Dependable and Adaptive Distributed Systems*, pages 691–696, Apr. 2006.

[93] O. Rütti, P. T. Wojciechowski, and A. Schiper. Structural and algorithmic issues of dynamic protocol update. In *Proceedings of IPDPS '06: the 20th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2006.

[94] *SAMOA*. `http://lsrwww.epfl.ch/samoa/`.

[95] A. Schiper. Dynamic group communication. *ACM Distributed Computing*, 18(5):359–374, Apr. 2006.

[96] P. Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of POPL '01: the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2001.

[97] P. Sewell, P. T. Wojciechowski, and B. C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages*, volume 1686 of *LNCS*, pages 1–31. Springer, Oct. 1999.

[98] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of PODCS '95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1995.

[99] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts, Sixth Edition*. John Wiley & Sons, Inc, 2002.

[100] N. Sridhar, S. M. Pike, and B. W. Weide. Dynamic module replacement in distributed protocols. In *Proceedings of ICDCS '03: the 23rd International Conference on Distributed Computing Systems*, May 2003.

[101] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *Proceedings of POPL '05: the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2005.

[102] V. Tanasescu and P. T. Wojciechowski. Role-based declarative synchronization for reconfigurable systems. In *Proceedings of PADL '05: the 7th International Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *LNCS*, pages 52–66. Springer, Jan. 2005.

[103] A. S. Tanenbaum. *Modern Operating Systems, Second Edition*. Prentice Hall, Englewood Cliff, NJ, 2001.

[104] J. D. Ullman. *Elements of ML Programming*. Prentice-Hall, 2nd (ML97) edition, 1997.

[105] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software Practice & Experience*, 28(9):963–979, July 1998.

[106] J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A semantic framework for designer transactions. In *Proceedings of ESOP '04: the 13th European Symposium on Programming*, volume 2986 of *LNCS*. Springer, March/April 2004.

[107] C. Walton, D. Kirli, and S. Gilmore. An abstract machine model of dynamic module replacement. *Future Generation Computer Systems*, 16:793–808, May 2000.

[108] G. Weikum and G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.

[109] J. M. Wing, M. Faehndrich, J. G. Morrisett, and S. Nettles. Extensions to Standard ML to support transactions. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[110] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[111] P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, University of Cambridge, June 2000. Also published as Technical Report 492, Computer Laboratory, University of Cambridge.

[112] P. T. Wojciechowski. Concurrency combinators for declarative synchronization. In *Proceedings of APLAS '04: the 2nd Asian Symposium on Programming Languages and Systems*, volume 3302 of *LNCS*, pages 163–178. Springer, Nov. 2004.

[113] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. Technical Report IC-2004-104, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Dec. 2004.

[114] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proceedings of PPDP '05: the 7th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.

[115] P. T. Wojciechowski. A class-based object calculus of dynamic binding: Reduction and properties. In *Proceedings of SC '06: the 5th IFIP International Symposium on Software Composition (co-located with ETAPS '06)*, volume 4089 of *LNCS*, pages 131–146. Springer, March 2006.

[116] P. T. Wojciechowski, S. Mena, and A. Schiper. Semantics of protocol modules composition and interaction. In F. Arbab and C. Talcott, editors, *Proceedings of COORDINATION '02: the fifth International Conference on Coordination Models and Languages*, volume 2315 of *LNCS*, pages 389–404. Springer, Apr. 2002.

[117] P. T. Wojciechowski, S. Mena, and A. Schiper. Semantics of protocol modules composition and interaction. Technical Report IC-2002-2, School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Feb. 2002.

[118] P. T. Wojciechowski and O. Rütti. On correctness of dynamic protocol update. In *Proceedings of FMOODS '05: the 7th IFIP Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 3535 of *LNCS*, pages 275–289. Springer, June 2005.

[119] P. T. Wojciechowski, O. Rütti, and A. Schiper. SAMOA: A framework for a synchronisation-augmented microprotocol approach. In *Proceedings of IPDPS '04: the 18th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2004.

[120] P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2):42–52, April–June 2000.

[121] G. T. Wong, M. A. Hiltunen, and R. D. Schlichting. A configurable and extensible transport protocol. In *INFOCOM '01: the 20th IEEE Conference on Computer Communications*, Apr. 2001.

[122] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[123] *X-kernel*. `http://www.cs.arizona.edu/xkernel/`.

Index

Streszczenie

Paweł T. Wojciechowski

Projekt języka operacji atomowych, deklaratywnej synchronizacji i dynamicznej aktualizacji w systemach komunikacyjnych

Obecnie występuje rosnące zainteresowanie projektowaniem nowych języków programowania, które łączą takie cechy, jak współbieżność (lub równoległość) oraz możliwość dodawania nowego kodu w trakcie działania programu. Dzięki tym cechom możliwości procesorów wielordzeniowych mogą być lepiej wykorzystane, pozwalając przy tym na dynamiczną aktualizację oprogramowania. Przykładowymi aplikacjami są usługi rozproszone, które muszą być dostępne bez przerwy, np. finansowe i telekomunikacyjne, rezerwacje lotów oraz kontrola ruchu lotniczego. Zatrzymanie usług świadczonych „non-stop" powoduje straty finansowe; może także spowodować zagrożenie bezpieczeństwa. Dlatego też usługodawcy muszą móc naprawić, zaktualizować lub rozszerzyć swoje systemy z minimalną ingerencją w dostępność usługi. Dające się dynamicznie aktualizować systemy rozproszone mogą być implementowane z użyciem popularnych języków programowania, które pozwalają na dynamiczne ładowanie i łączenie komponentów kodu. Jednakże, aby usprawnić programowanie oraz zagwarantować niezawodność, potrzebne są nowe abstrakcje programistyczne. Przykładem są konstrukcje do synchronizacji różnych operacji wejścia/wyjścia, wykonywanych lokalnie w maszynie lub globalnie w systemie rozproszonym.

W tej rozprawie zaprojektowano nowe konstrukcje językowe i algorytmy uzyskiwania atomowości, deklaratywnej synchronizacji oraz dynamicznej aktualizacji protokołów. Mogą one służyć do budowy systemów komunikacyjnych z modularnych protokołów, które można zastępować dynamicznie.

- *Atomowość* (niepodzielność) gwarantuje, że zbiór operacji wykonywanych w miejscu sieciowym (maszynie) może być brany pod uwagę jako pojedyncza jednostka obliczeniowa, niezależnie od jakichkolwiek innych operacji wykonywanych współbieżnie.

- *Deklaratywna synchronizacja* oznacza sposób implementacji sterowania różnego rodzaju współbieżnymi akcjami lub zdarzeniami w systemie, który polega na zdefiniowaniu polityki synchronizacji (takiej jak atomowość). Definiuje się ją w formie zbioru reguł, oddzielnie od kodu komponentów. Takie podejście umożliwia ponowne użycie komponentów protokołów w różnych stosach protokołów oraz ułatwia ich dynamiczną zamianę.

- *Dynamiczna aktualizacja protokołów* oznacza transparentną zamianę protokołów w trakcie działania systemu w taki sposób, że korzystanie z usług implementowanych przez te protokoły nie jest narażone na błędy. Jednoczesna dynamiczna zamiana komponentów protokołów zlokalizowanych w różnych miejscach sieciowych odbywa się pod kontrolą *algorytmów przełączania*.

Rozprawa ma następującą strukturę. Rozpoczęto od przedyskutowania motywacji i kontrybucji. Następnie opisano algorytmy wersjonowania przeznaczone do sterowania współbieżnością w zadaniach atomowych. W kolejnym rozdziale zaprojektowano rachunek (ang. *calculus*) zadań atomowych, tj. transakcji atomowych bez odtwarzania stanu, mogących mieć efekty wejścia/wyjścia. Rachunek ten wyposażony jest w system typów do statycznej weryfikacji danych wymaganych przez dynamiczne wersjonowanie. System typów gwarantuje, że proponowane w rachunku konstrukcje programistyczne są używane w sposób prawidłowy. Następnie opisano dwa różne podejścia do synchronizacji deklaratywnej: (1) rachunek kombinatorów współbieżności z opartą na typach weryfikacją spełnialności kombinatorów (co daje gwarancję ich poprawnego zastosowania) oraz (2) język ograniczeń (ang. *constraint language*) dla modelu synchronizacji opartego na rolach. W kolejnym rozdziale opisano model dynamicznej aktualizacji protokołów oraz podano dwa przykładowe algorytmy przełączania protokołów, mające określone pożądane własności. W ostatnim rozdziale zaprojektowano oparty na klasach obiektowy rachunek wiązań dynamicznych. Rachunek posłużył do pokazania zastosowań mechanizmów synchronizacji opisanych w tej książce (tj. zadań atomowych i kombinatorów współbieżności) przy dynamicznym wiązaniu obiektów. W dodatku zamieszczono formalne dowody poprawności szeregu twierdzeń wykazujących trafność (poprawność) systemu typów (ang. *type soundness*) dla rachunku zadań atomowych, w tym dowód dynamicznej poprawności przykładowego algorytmu wersjonowania.