This is to certify that the
dissertation entitled

Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages

presented by

Karl Reimer Behrends

has been accepted towards fulfillment
of the requirements for the

Ph.D. degree in Computer Science and Engineering

R. E. K. Stirewalt
Major Professor's Signature

12/11/2003
Date

Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages

By

Karl Reimer Behrends

A Dissertation

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

Department of Computer Science and Engineering

2003

ABSTRACT

DESIGNING AND IMPLEMENTING A MODEL OF SYNCHRONIZATION CONTRACTS IN OBJECT-ORIENTED LANGUAGES

By

Karl Reimer Behrends

This thesis describes the design and implementation of a synchronization mechanism, called the universe model, that is based on declaratively specified contractual relationships between client and supplier components. These *synchronization contracts* not only specify basic mutual exclusion and semantic predicates, but also explicit logical composition of those basic contracts by boolean operators, and implicit composition to recombine contracts that have been decomposed in order to modularize them. Such contracts are negotiated implicitly and transparently by a runtime system that observes the abstract state of the program, instead of using explicit procedural synchronization primitives. We present a classification scheme for synchronization contracts, and describe how they can be added to existing object-oriented languages by means of a simple declarative language that expresses these contracts. We also present an implementation of a runtime system that negotiates these synchronization contracts: It draws upon ideas from garbage collection and distributed databases, and adapts them to negotiate synchronization contracts in a fash-

ion that avoids deadlocks and starvation where feasible. While the resulting algorithms are complex, we have implemented them and present empirical validation to show that this implementation is efficient.

TABLE OF CONTENTS

List of Tables

LIST OF FIGURES

Chapter 1

Introduction

Applications in which multiple threads operate on shared data are notoriously difficult to design. Without proper synchronization, concurrent access to shared objects can lead to race conditions [NM92], and incorrect synchronization logic can lead to starvation and deadlock. To address these concerns, we and others have looked to extend and articulate Meyer's *design by contract* method to raise the level of abstraction with which to design such systems (cf. [Mey97, BJPW99, BS00, SBD03]). Contracts used for this purpose are called *synchronization contracts* because they specify and constrain the legal sequences of interaction among objects and processes[1] in a concurrent system. For example, in a client–server system with one server and multiple clients, a synchronization contract could assert that a client can perform a sequence of server operations without interference by other clients. Our *universe model* [BS00, SBD03] is a powerful model of synchronization contracts that integrates well with existing object-oriented languages. A key concept in this

[1] We shall henceforth use the term process rather than thread, as the former is the term used more frequently by other researchers. Throughout this thesis we treat these terms as synonyms.

model is a run-time system that *dynamically negotiates* contracts among multiple groups of objects by scheduling processes so as to guarantee mutual exclusion while attempting to avoid starvation and avoid or recover from deadlock situations. This thesis describes the design of a programming model for such contracts, the algorithms used to implement run-time negotiation for them, documents extensions for the Eiffel programming language to use that model, and provides evidence for its scalability.

1.1 Synchronization Contracts

The term *contract* refers to a formal agreement between client and supplier modules. Beugnard and others [BJPW99] identify a taxonomy of different kinds of contracts, the most familiar type being *behavioral contracts*, which include the pre- and post-condition features of Eiffel [Mey92]. Behavioral contracts are specified in the interface of a supplier module, and they spell out the module's rights and responsibilities. In the design-by-contract method, each module is designed to assume the rights and guarantee the responsibilities of its contract, and the designer must *verify* that a client guarantees the rights assumed by any supplier the client uses. Meyer shows how such *contract-aware modules* are dramatically smaller and simpler than their contract-unaware counterparts [Mey97]. For example, the former will not contain code to check operation pre-conditions, as the designer will have verified that clients only invoke the operation when the pre-condition is satisfied.

Synchronization contracts afford a similar simplicity by allowing a designer to declaratively specify constraints on concurrent interaction in lieu of programming low-level synchronization mechanisms and protocols. Unlike behavioral contracts, synchronization con-

tracts are not verified at design time, but rather are *negotiated* at run time. In Java, for example, when multiple client objects (running in different threads) invoke a synchronized method on the same supplier object, the clients negotiate for exclusive access to the supplier. The mechanism used to support this negotiation is a simple mutex, which each client will attempt to lock before entering the method, but neither the client modules nor the supplier module contains code for acquiring and releasing this mutex.

More powerful synchronization contracts employ more powerful synchronization mechanisms. Consider, for example, a contract that guarantees a client exclusive access to multiple suppliers. To prevent undesirable phenomena, such as starvation and deadlock, clients and suppliers must negotiate this contract using a protocol, such as one of the common two-phase protocols [RSL78, FR85, HZ92, FHRT92]. The logic for such protocols is not localized to a single client or supplier module. Thus the ability to specify this behavior declaratively, via a contract, should simplify (as well as decouple) the implementation of client and supplier modules. We believe that integrating ever more powerful models of synchronization contracts into programming languages will dramatically simplify the development and improve the reliability of multi-threaded shared-memory systems.

1.2 The Universe Model

The universe model supports a powerful class of synchronization contracts. The most basic forms are *exclusion contracts*, which guarantee exclusive access to a supplier S from a client C, and *conditional contracts*, which guarantee that C has exclusive access to S once some condition in S is satisfied. The model also supports *composite contracts*, which

combine exclusion and conditional contracts by conjunction and disjunction. Composite contracts are used, for example, to guarantee a client exclusive access to multiple suppliers. Moreover, composite contracts may arise implicitly, as when one module m_1 has an exclusion contract with a supplier module m_2, which itself has an exclusion contract with another supplier m_3. This transitivity of contract dependencies is essential for guaranteeing mutual exclusion without violating information hiding, especially in hierarchically layered systems [BS00]. Finally, the model also supports *parameterized contracts*, which prefix a contract with an enabling predicate that may refer to complex state conditions (i.e., the parameters) in the client. Parameterized contracts are used to specify how a client's needs change according to changes in the state of the client. At runtime, changes to these parameters implicitly trigger a renegotiation of the contract.

Because the underlying mechanisms to negotiate individual exclusion and conditional contracts are well understood, support for these basic contracts is common in modern object-oriented languages. By contrast, composite contracts have yet to gain widespread adoption in general purpose programming languages[2], presumably because the synchronization mechanisms required to dynamically negotiate them are still the subject of current research. This thesis contributes the design of a model for the general composition of synchronization contracts, and an algorithm that uses a combination of synchronization mechanisms and protocols to dynamically negotiate these composite contracts while avoiding starvation and avoiding or recovering from a large class of deadlocks. To validate the algorithm, we extended the Eiffel language with primitives to declare universe-model contracts,

[2] A counterexample is [FP97].

and we implemented a compiler and run-time system for this extended language.

1.3 Thesis Overview

The remainder of the thesis is organized as follows. First, we survey existing research in the area of concurrency, and establish the relationship between synchronization mechanisms and synchronization contracts (Chapter 2). Second, we formally introduce the universe model and its contract-specification language (Chapter 3). A major portion of the thesis (Chapter 4 through Chapter 7) describes our algorithm to dynamically negotiate contracts and schedule processes. This algorithm borrows and extends ideas from reference-counting garbage collection [BAL+01] and combines deadlock avoidance and recovery mechanisms [RSL78, HZ92],in order to negotiate contracts of the power expressible in the universe model. For clarity of exposition, we first present the algorithm for a restricted class of contracts, namely exclusion contracts and their explicit and implicit composites, and a negotiation protocol that is starvation free and that avoids a limited class of deadlocks. We then successively refine this algorithm to include a deadlock-recovery heuristic, which expands the class of preventable deadlocks, and to handle the (more general) conditional contracts. Chapter 8 then describes a set of extensions to the Eiffel programming language that make the semantics of the universe model available to the programmer. Finally, we demonstrate that the algorithm, as implemented in our extended Eiffel implementation, operates with reasonable efficiency (Chapter 9) and conclude by comparing the universe model to alternative mechanisms that support synchronization contracts and discussing follow-on open research questions (Chapter 10).

Chapter 2

State of the Art

Before defining the universe model in detail, we first survey the state of the art of current research with respect to concurrency management. Fundamental for the study of concurrency management is the distinction between the two most common execution models for concurrent systems, shared memory and message passing environments. These two models differ both in their implementation and the programming paradigms that are available in each model. In the shared memory model, concurrent processes access shared data and this access must be regulated to avoid race conditions [NM92] (Section 2.1). In the message passing model, each concurrent process exists in a separate dataspace and communicates with other processes only via the sending and receiving of messages. Message passing in a concurrent system is inherently non-deterministic and must be coordinated to avoid unpredictable behavior (Section 2.2). A variety of approaches have been developed based on these paradigmatic differences. Both execution models, however, assign client and server (or supplier) roles to objects and resources, where clients access servers (concurrently) to request services. This provides another dimension along which concurrent programming

models can vary, focusing either on the client or the supplier as being responsible for synchronization (Section 2.3)

A major challenge with either approach is to create high-level models that support the programming of reliable concurrent systems. A key element in the design of such high-level approaches is the treatment of *multi-step transactions*, which perform multiple operations on one or more server objects atomically. Language support for multi-step transactions is scarce (Section 2.4) and where available, introduces modularity problems, such as violation of information hiding, in hierarchically structured software designs (Section 2.5). We conclude the survey of current research with an overview of existing synchronization mechanisms that support multi-step transactions, and classify them according to their capabilities (Section 2.6).

2.1 Synchronization in Shared Memory Systems

The shared memory model of parallel programs assumes that processes share a common memory area. Processes communicate by writing to and reading from a shared object in that area. While communication through shared data structures is an essential feature of the shared memory model, it is also an inherent problem. Consider the code in Figure 1, which depicts two processes, P1 and P2. When P1 and P2 are running concurrently, instructions can be executed in a variety of orders. The sequence ⟨1⟩, ⟨2⟩, ⟨3⟩, ⟨4⟩ is just as possible as ⟨1⟩, ⟨3⟩, ⟨2⟩, ⟨4⟩ or ⟨3⟩, ⟨1⟩, ⟨2⟩, ⟨4⟩. Each version will result in the variables `x` and `y` containing different values. Also, statements ⟨1⟩ and ⟨3⟩ can be executed simultaneously, which makes the contents of `t` unpredictable. In other words, P1 and P2 *interfere* with each

```
process P1                      process P1
  t := x * c1   ⟨1⟩               t := y * c2   ⟨3⟩
  x := t        ⟨2⟩               y := t        ⟨4⟩
end process                     end process
```

Figure 1: Two concurrent processes

other.

Interference between processes occurs due to race conditions [NM92], which can take the form of either *data races*, where two processes access the same data item simultaneously, or *general races*, where the execution of a process Q concurrently with another process P alters the behavior of P. For example, in Figure 1, both data races and general races can occur. Program correctness in shared memory systems is therefore generally predicated on the concept of *non-interference* [OG76], which requires programming mechanisms that guarantee non-interference by eliminating race conditions.

Traditionally, to guarantee non-interference, one must identify *critical sections* of code, where processes might interfere with one another, and then use *mutual exclusion mechanisms*, such as semaphores [Dij68], to protect these critical sections. Most concurrent programming languages provide higher-level constructs, such as monitors [Hoa74], which automatically lock and unlock a module on procedure entry and exit, to describe mutual exclusion mechanisms. One typically says that a shared resource or object has been *locked* or *acquired* by a process when other processes are excluded from accessing it.

Semaphores and monitors supply all the mechanisms needed to ensure non-interference. The real problem, however, is the correct identification of critical sections

and the generation of correct mutual exclusion code. If critical sections are misidentified, then shared resources can be accessed concurrently, leading to unpredictable errors. Furthermore, if mutual exclusion for more than one shared resource is needed, incorrect mutual exclusion code may introduce deadlock or starvation [Tan92]. Most high-level approaches that employ a shared memory model (including the universe model) employ mechanisms to reduce or eliminate the potential for error. In particular, all data races can be eliminated automatically by ensuring that a shared object is locked before it is accessed or modified. Automatic elimination of data races is thus a minimum non-interference guarantee that any safe synchronization mechanism can offer, though several elect not to follow the above approach for efficiency reasons [GJSB96] or to allow intra-object parallelism [CH74, Mit95].

However, this assurance cannot automatically be extended to general races. General races occur primarily during *multi-step transactions*, which consist of sequences of service invocations by and among multiple shared objects. We use the word transaction to suggest that the sequence should either run to completion or not at all; that is, we want to remove the possibility that service requests from other concurrent processes can interfere with the sequence.

Many programming languages allow the programmer to guard transactions with a mechanism for locking an object for the duration of such a transaction. However, even if one requires that an object is locked before it is accessed (as in [Mey93]), there is no guarantee that the lock is held throughout the acquisition. For instance, process P1 in the above example could be rewritten as follows, where the `synchronized` statement locks `t` for the duration of the enclosed block:

```
process P1
  synchronized t do
    t := x * c1
  end
  synchronized t do
    x := t
  end
end process
```

Observe that the code, after having been augmented with synchronization instructions, will not access the shared object `t` without locking it first; however, the program is just as vulnerable to interference as the original. While this defect is easy to detect in such a brief segment of code, common programming practice can easily hide it.

Assume, for example, that `x`, `t`, and `c1` are matrices. In such a case, the matrix operations are often performed by subroutine calls, and temporary results are stored in an intermediate variable to avoid unnecessary copying of a two-dimensional array. Suppose that functions `multiply` and `assign_from_tmp` reference an intermediate variable (called `t`), which is global (i.e., not local to these functions). The correct code to avoid non-interference would then look as follows:

```
process P1
  synchronized t do
    call multiply(x, c1)
    call assign_from_tmp(x)
```

end

end process

To write the correct synchronization code requires that the programmer inspect the internals of `multiply` and `assign_from_tmp` to derive that `t` needs to be locked for the duration of the transaction. This obligation is easy to overlook and its omission is a clear violation of basic information hiding, because it exposes `t`. However, moving the `synchronized` instruction inside the subroutines for proper information hiding would allow other processes to interfere with the operation.

In summary, the shared memory model is prone to interference due to race conditions. Race conditions must be eliminated through mutual exclusion mechanisms, which cannot always guarantee the absence of general races and may violate information hiding.

2.2 Message Passing Systems

In a message passing system, each process has its own memory area, disjoint from all the others. Processes communicate by sending messages to one another. Having exactly one process per memory area automatically eliminates all data races and is considered the major advantage of the message passing model [Hoa85]. Also, convenient formalisms exist that make it easy to reason about the correctness of such programs [Hoa85, Mil80]. However, if one wishes to harness the message passing model for concurrent programming by providing general purpose programming mechanisms, a number of new concurrency-related problems arise that do not exist in the shared memory model. In this section, we

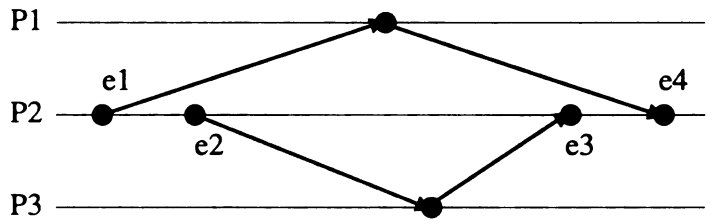

Figure 2: Message Ordering

explore these problems and possible solutions to them.

The first problem is that of a copying overhead. When two processes want to access the same data structure, the data must be copied. In the case of complex data structures, the messages sent between processes can become so large that the communication overhead dwarfs the actual computation. Since memory access speed has not increased at the same rate as CPU speed, this can be a concern for time-critical applications.

The second problem is that of *message ordering*. Where the shared memory model has a non-determinism problem when it comes to accessing shared data, the corresponding problem in the message passing model is message ordering for computations that involve several processes. Consider the time diagram in Figure 2. Horizontal lines represent processes, arrows represent messages being sent, and the filled circles represent the events that occur when messages are received. An event being depicted to the right of another event means that it occurred at a later time. Whether event *e3* occurs before event *e4* (as depicted in the figure) or vice versa depends solely on the order in which the corresponding messages arrive. The underlying problem is that message passing is an asynchronous operation (meaning that messages initiate an independent thread of control in the receiver, while the sender's thread continues in parallel), whereas procedure calls in the shared memory model

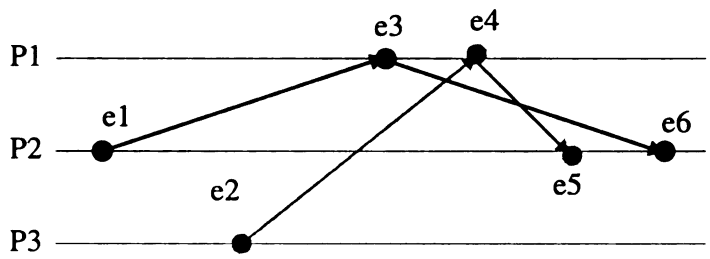

Figure 3: Causality

are synchronous operations.

Non-deterministic message arrival can cause problems when it violates *causality*. Intuitively, causality violations occur when an event e' depends on the results of an event e, but a process observes and acts upon the effects of e' before it observes and acts upon the effects of e (a formal definition can be found in [SM94]). For example, in Figure 3, process P2 sends a message (*e1* to *e3*) to inquire about P1's state. P1 then sends a reply message (*e3* to *e6*) with the required data. By the time the message arrives (*e6*), P1 has received another message (*e4*), which has altered the state, so that the information is out of date. Here, *e4* depends on the results of *e3*, but P2 observes and acts upon the results of *e4* in *e5*, before it observes and acts upon the results of *e3* in *e6*. This is a causality violation.

The third problem is that because of the non-determinism of message passing, it can be difficult to obtain a *globally consistent view* of the state of more than one process (also called a *distributed global snapshot*). Specifically, if we have processes $P_1, \ldots, P_n$, with states $S_1(t), \ldots, S_n(t)$ expressed as a function of the time t, then we may wish to compute a function f that depends on several of the $S_i(t)$. Because message transit takes an unknown, but non-zero amount of time, the values of the $S_i(t)$ will typically be gathered at different

times. If any state changes occur between the beginning and the end of the computation (a violation of causality), then the result of f may be incorrect. The general problem of computations that depend on the state of more than one process is sufficiently complex to have spawned several independent research areas [Mat93]. Specifically, the case of obtaining a consistent global snapshot has been shown to be inherently non-trivial [CT90].

Both the second and the third problem can be solved by having a message passing mechanism that preserves causality. Unfortunately, the known general purpose algorithms [Fid88, Mat89] that preserve causality use a vector of timestamps, containing one timestamp per process, and require a worst-case $O(n)$ overhead per message, where n is the number of processes. While there also exist a number of more efficient algorithms for the area of distributed discrete-event simulations [CM81, Jef85] that preserve causality, they require that each event carry a global timestamp. This requirement limits their applicability to other domains.

Despite these problems, the message passing model has the advantage that it is very amenable to being used in combination with formal methods [BB89, Mil80]. Even when a full formalization is not possible, event-based schemes are generally so well understood that they can be analyzed with reasonable ease by modeling them with statecharts [Har87]. However, based on the existing research we conjecture that a fully automated way to handle causality-related problems may incur a substantial overhead that is linear in the number of processes.

2.3 High-Level Synchronization Paradigms

Having discussed the underlying execution models, we now turn to higher-level paradigms that provide abstractions to deal with these execution models in a more convenient fashion. Object-oriented techniques fall either in the category of *passive objects* or *active objects*. In the active object model [Nie95], every object is driven by an autonomous process, communicating with other objects via message passing. Since it is often not practical to have one separate thread of control for each object in a large system, most approaches actually have one active object offering services and controlling several non-active objects that it communicates with via synchronous communication [Car90, YT87]. In the passive object model, objects do not embody processes. Instead, processes are threads of control that exist outside the object system and that call methods of shared objects directly. When one relates these models to message passing and shared memory approaches, active objects are the object-oriented equivalent of the message passing model, whereas passive objects implement the shared memory model.

Both active and passive object systems distinguish between clients and servers. A server object is a provider of services, which can be invoked by one or more clients. In the passive object model, services are invoked by procedure or method calls; in the active object model, services are invoked by sending a message to the server. Synchronization for a client-server interaction can either be performed by the server or by the client, and synchronization mechanisms can vary considerably depending on which side is responsible for synchronization [Bri72, CH74, FP97, Hoa74, Hol99, Mey93].

The *design by contract* approach prefers to call servers suppliers, but server is the es-

tablished term for concurrent systems. We will therefore use the term *server* in the context of synchronization mechanisms as an object or process that interacts concurrently with a client, and the term *supplier* to emphasize the role as the provider of services in a contractual context.

2.3.1 Server-side Synchronization for Passive Objects

Among the first high-level mechanisms for shared memory systems was the concept of a monitor [Hoa74], developed by Hoare and Brinch Hansen. A monitor is strictly a server-side synchronization mechanism that is associated with each shared object: each time a method in the shared object is invoked, the object is locked; upon return from the called method, the object is unlocked. Monitors incorporate the concept of *condition variables*, which embody an abstract condition. Processes can wait for the condition to become true or signal that the condition has become true.

Monitors localize synchronization within the shared object that is being accessed and automatically eliminate all data races. However, they cannot easily handle multi-step transactions. Specifically, a monitor cannot be locked for a sequence of service invocations or manage transactions that need to invoke services on more than one shared object without risk of interference. While this problem can be handled for transactions by migrating client code into the server (where the code becomes non-interruptible), such migration inflates the module interface of server objects and cannot solve the problem where a client needs to lock multiple servers.

Subsequent approaches attempted to correct the problem of dealing with multi-step

transactions. Path expressions [CH74] allow a programmer to specify as part of the interface of a shared object all legal combinations of service invocations. Such combinations include both sequences of service invocations (thus handling multi-step transactions) and which services can be invoked concurrently (increasing parallelism when compared to monitors). The disadvantage of path expressions is that an error in the specification of a path expression can create data races. Also, with respect to multi-step transactions, a path expression has to anticipate all possible sequences of service invocations in order for a client to be able to actually use them. Should the actual access pattern by a client and the path expression disagree, this error is not necessarily detectable. Moreover, path expressions cannot handle transactions that involve multiple servers.

A recent approach that attempts to deal with multi-step transactions on the server side is the concept of *synchronization rings* [Hol99]. Synchronization rings are wrapper objects around server objects that are separate from the server proper, but that can intercept service invocations on one or more servers for synchronization purposes. Monitors can be simulated by having one synchronization ring for each shared object, which locks the object upon the beginning of a service invocation and unlocks it at the end. Unlike a monitor, however, a synchronization ring can intercept service invocations for more than one shared object and can synchronize access for entire multi-step transactions in a fashion similar to path expressions. The disadvantage of synchronization rings is that they cannot guarantee the absence of data races in general, and (like path expressions) need to anticipate and duplicate all possible sequences of service invocations by clients.

2.3.2 Client-side Synchronization for Passive Objects

Given the inherent difficulty of modeling synchronization for multi-step transactions on the server side, a fair amount of research has focused on client-side synchronization. One of the most powerful early approaches was that of *conditional critical regions* [Bri72]. While never implemented in their original form, conditional critical regions formed the prototype for more advanced approaches, as well as proof techniques such as the Owicki-Gries model [OG76].

A conditional critical region is a statement of the form

lock R **when** C **do** S **end**.

This statement declares that as soon as the shared object `R` is available *and* condition `C` becomes true, `R` is locked and statement `S` is executed. By ensuring that R can only be accessed from within a conditional critical region, data races can be avoided. Obviously, conditional critical regions can handle multi-step transactions of arbitrary length without additional effort, as long as they use only the single server `R` during the transaction.

An obvious extension to conditional critical regions is to let `R` be a set of shared objects, which are locked atomically as a group, rather than `R` being a single resource. In this extension, all resources in `R` have to be locked *and* `C` has to be true before `S` is executed. Such *extended conditional critical regions* are the core concept upon which modern client-side synchronization mechanisms such as SCOOP [Mey93, Mey97] and the Concurrent Extensions of Eiffel (CEE) [JP93] are built[1]. CEE implements a **holdif** statement that has

[1]While it has been argued whether SCOOP and CEE use active and passive objects, both use mechanisms that translate easily to general passive object schemes.

the semantics of extended conditional critical regions. SCOOP implements extended conditional critical regions as procedures: each argument passed to such a procedure that is a shared object is considered to be part of R, and the precondition of the procedure functions as the equivalent of C (preconditions are explicitly written in Eiffel, the language on which SCOOP is based). The programming language Sather [Omo91] extended the concept of conditional critical regions even further, allowing the specification of several alternative sets of resources, where the acquisition of one of those alternative sets was sufficient to enter the critical region. Thus, one could express conditions where exclusive access to either one set of resources or another (or yet another, and so forth) was required. Sather's locking mechanism is very powerful, but the actual implementation allowed for data races on shared objects, if the programmer accidentally forget to include them in the lock statement.

In general, while client-side mechanisms can handle multi-step transactions more easily than server-side mechanisms, they are frequently criticized because synchronization code needs to be replicated in each client, rather than being encapsulated in a single shared object [Blo79]. A large number of client-side synchronization mechanisms also permit data races (notably Java [Han99]).

An inherent problem in client-side synchronization mechanisms is that a client may specify that more than one server object is to be locked atomically, and that naive implementations of the locking of multiple shared objects may lead to deadlock. For example, if a process $P1$ locks a shared object A first, and another shared object B second, while another process P2 locks B first and A second, it is then possible that $P1$ has acquired A, P2 has acquired B, at which point $P1$ cannot acquire B nor can P2 acquire A, and the two processes are deadlocked. To solve the problem of acquiring multiple shared objects, one

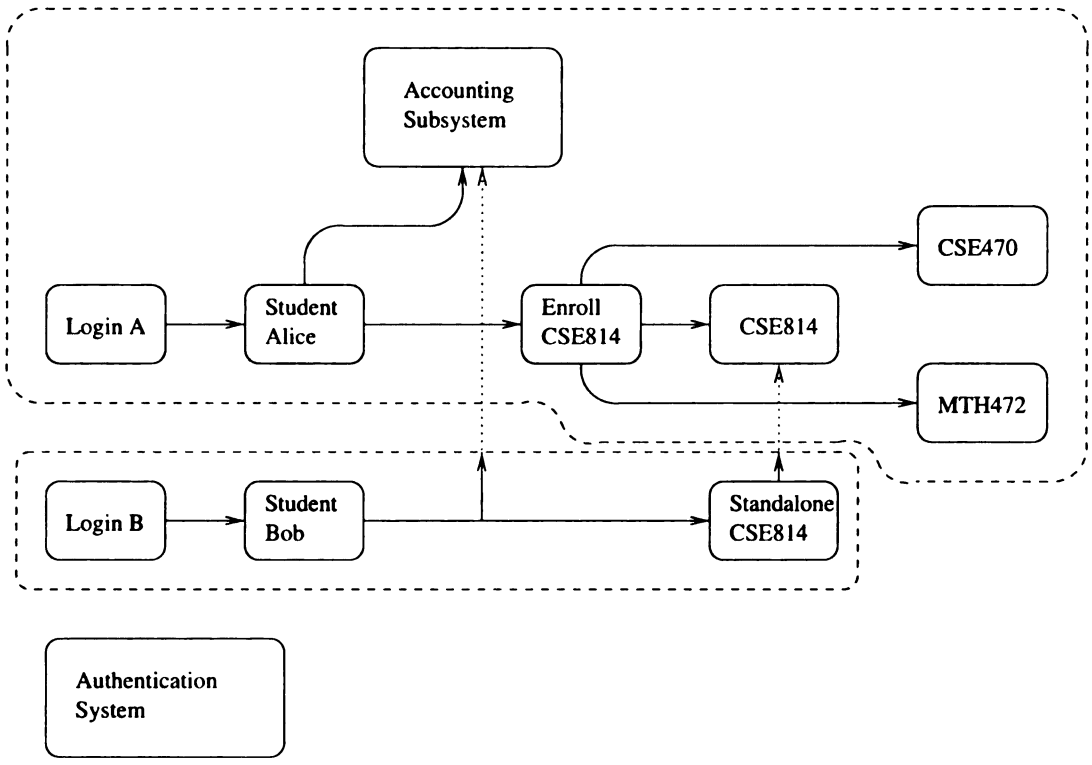

Figure 4: Enrollment Example

has therefore to construct a procedure for doing so that avoids deadlocks.

The problem of locking multiple shared objects occurs commonly in practice. Consider, for example, Figure 4, taken from [BS00]. It depicts part of the enrollment system of a fictional university. Boxes indicate objects in a passive object system, and arrows indicate that one object uses the services of another. Alice's ongoing transaction involves no fewer than five shared objects (the accounting subsystem, EnrollCSE814, CSE814, CSE470, MTH472). Bob's transaction, when it is his turn, will require at least three (the accounting subsystem, StandaloneCSE814, CSE814). The objects EnrollCSE814 and StandaloneCSE814 are distinct because they implement different synchronization contracts and provide different methods (the former requiring access to a course and its prerequisites, the

latter requiring access only to the course). The enrollment scenario therefore requires the locking or acquisition of multiple shared objects.

2.3.3 Active Object Approaches

Because the focus of this thesis is on passive object schemes, we only briefly discuss the most interesting active object synchronization schemes. One of the oldest approaches to synchronization is Hoare's concept of *Communicating Sequential Processes*, or CSP [Hoa78]. It has been the basis both for programming languages such as OCCAM-2 [INM88] and specification languages such as LOTOS [BB89]. Its influence can also be seen in the design of Ada's Rendezvous mechanism.

CSP combines concurrency control (such as parallel and alternate execution) and input and output (across message channels) as primary programming language constructs based on the message passing model, facilitating the development of concurrent programs at a higher level of abstraction. In addition, CSP was the first attempt at a process algebra. Process algebras, such as CSP, CCS [Mil80], or ACP [BK85], provide algebraic languages for the specification of processes and calculi for reasoning about their behavior.

More recently, the field of *coordination languages* [PA98] evolved from the idea that computation and coordination can be separated in active objects. Coordination, in its most general form, "is the process of building [concurrent] programs by gluing together active pieces" [CG92]. Hoare's Communicating Sequential Processes can be viewed as a model for a coordination language, though Linda [GCCC85] is typically considered the earliest genuine coordination language.

```
process philosopher(i)
  loop
    think;
    in("room ticket");
    in("fork", i);
    in("fork", i mod 5 + 1);
    eat;
    out("fork", i);
    out("fork", i mod 5 + 1);
    out("room ticket");
  end
end
```

```
process main
  for i := 1 to 5 do
    out("fork", i);
    eval(philosopher(i));
    if i < 5 then
      out("room ticket", i);
    end
  end
end
```

Figure 5: Dining Philosophers in Linda

Among coordination languages, we distinguish between data-driven and control-driven languages. Linda is the prototypical example of the former: all processes are separate entities, but have access to a shared data space, called a *tuple space*. The two basic primitives in Linda allow the programmer to insert an item (a tuple) into the tuple space, or to retrieve an item that matches a certain pattern. Figure 5 shows the classical dining philosophers problem in Linda.

Here, the `out` primitive inserts a tuple such as `("fork", 1)` into the shared tuple space. The `in` primitive retrieves a tuple from the tuple space; should there be no matching tuple in the space, the process blocks until another process inserts one. The main advantage of a shared tuple space is that as a central authority on concurrency management, it reduces the causality and global view problems of active object systems.

This solution to the dining philosophers problem is already much more straightforward than the traditional ones based on mutual exclusion (such as in [Tan92]). But the more

general advantage of a coordination language is that processes can be designed separately from one another, interfacing through separate coordination interfaces.

Unlike data-based coordination languages, control-based coordination languages do not know the concept of a shared data space. In control-based coordination languages, processes are black boxes, communicating with their environment via *input ports* and *output ports* defined in their interfaces. Channels between processes then create a set of producer-consumer relationships between output ports of producers and input ports of consumers. Examples of control-based coordination languages are PCL [TGC93], the POLYLITH Software Bus [Pur94], or WRIGHT [AG97].

Of particular interest among the control-based variants are compositional coordination languages such as Strand [Pat90] or PCN [Fos96]. Rather than coordinating processes as black-box *modules*, these languages allow pieces of *functional* code to be composed to form more complex processes. Their major disadvantage is that they require programs to be written using functional or logical languages, since code with side-effects does not meet their requirements for composability.

Almost all coordination languages rely on the active object model. Like CSP, they share its advantages and shortcomings. Relying on data flow architectures, few coordination languages support coordinated multiple resource access natively (an exception is the synchronizer concept [FA93], which operates in a fashion similar to the synchronization rings discussed in Section 2.3.1).

2.4 Synchronization of Multi-step Transactions

In Section 2.1, we introduced the concept of a multi-step transaction as a sequence of service invocations by and among multiple objects to perform some task and noted that most languages guard transactions with a mechanism for *synchronizing* access to an object for the duration of such a transaction, such as Java's *synchronized* primitive or the *holdif* statement of CEE. For example, in Java, the statement `synchronized(ob) { ... }` directs the run-time system to execute an entire block of statements atomically with respect to the object `ob`. Consequently, the process may perform a multi-step transaction on `ob` without concern for interleaved invocations from other (concurrent) processes.

Unfortunately, a compiler cannot determine if a sequence of statements *should* execute as a multi-step transaction. Consequently, the proper use of such a feature cannot be checked by the compiler and therefore requires designer discipline to use correctly. Consider, for example, that the programmer forgot to encapsulate an intended multi-step transaction inside the `synchronized(ob) { ... }` construct. The program would compile without error, and the omission might go undetected for many runs until some unfortunate timing sequence leads to an undesired interference by another process.

In traditional programming languages, multi-step transactions and the participation of an operation in a multi-step transaction cannot be declared as part of a server module's interface. Instead, transactions must be implemented inside client code, which means that the client programmer must know that certain sequences of server operations must be protected in order to run atomically. Moreover, the use of features like `synchronized` violates the principle of information hiding (as we have seen in Section 2.1), because the programmer

may have to expose information about the implementation of the synchronized code to properly use this feature.

Features such as `synchronized` are symptomatic of the generally limited support [FP97] that popular programming languages provide for separation of concurrency concerns. The module interface of `ob` lists a set of services with no hint of constraints on the sequence of invocations of these services. Because interfaces cannot represent sequencing constraints in client-side synchronization schemes, there is no way for a compiler to detect the absence of a synchronized statement or to enforce the proper sequencing at run time. However, to avoid an error, a programmer must know to guard a sequence of `ob`-service invocations from interleaved `ob` calls by other concurrent processes.

2.5 Synchronization and Composability

Multi-step transactions are often not confined to accessing the services of only a single server, as seen in Figure 4 in Section 2.3.2. Instead, they may access multiple shared objects, all of which need to be locked for the duration of the transaction. We now discuss how the specification of transactions that access multiple shared objects affects the modularity and composability of code.

The archetypical example for transactions that access more than one shared object is the case of the dining philosophers, where each philosopher (as a multi-step transaction) will repeatedly pick up two forks, eat with them, and put the forks down again. Textbook solutions usually involve tricky semaphore manipulations, whereas the basic constraint of the system is straightforward: Whenever a philosopher eats, he requires access to both the

fork on his left-hand side and the fork on his right-hand side.

In a semi-formal notation, we might express this constraint as follows:

$$eating \Rightarrow left\_fork \wedge right\_fork$$

and write the code for a philosopher as follows, where `eat` implements the relevant multi-step transaction:

```
process philosopher
  loop
    think;
    eating := true;
    eat(left_fork, right_fork);
    eating := false;
  end
end
```

From this code and the accompanying constraint, a runtime system should be able to derive proper synchronization requirements. Now, compare how this brief, elegant requirement translates to an ad-hoc monitor-based solution in Figure 6, taken from [BA90].

From the standpoint of understandability, the solution in Figure 6 is inferior to the smaller and more transparent solution above. Not only does the former fail to capture the essence of the problem, but the salient synchronization information has been merged with the program code in a way that makes it close to impossible to identify concurrent interaction. Instead of abstracting and separating this information, the information is spread across

```
monitor Fork_Monitor
  Fork: array(0..4) of Integer range 0..2:=(others=>2);
  OK_to_eat: array(0..4) of Condition;

  procedure Take_Fork(I: Integer) is
  begin
    if (Fork(I) /= 2 then Wait(OK_to_Eat(I))); end if;
    Fork((I+1) mod 5) := Fork((I+1) mod 5)-1;
    Fork((I-1) mod 5) := Fork((I-1) mod 5)-1;
  end Take_Fork;

  procedure Release_Fork(I:Integer) is
  begin
    Fork(I+1) mod 5) := Fork((I+1) mod 5)+1;
    Fork(I-1) mod 5) := Fork((I-1) mod 5)+1;
    if Fork((I+1) mod 5)=2 then
      Signal(Ok_to_Eat((I+1) mod 5);
    end if;
    if Fork((I-1) mod 5)=2 then
      Signal(Ok_to_Eat((I-1) mod 5);
    end if;
  end Release_Fork;
end Fork_Monitor;
```

Figure 6: Dining Philosophers using Monitors

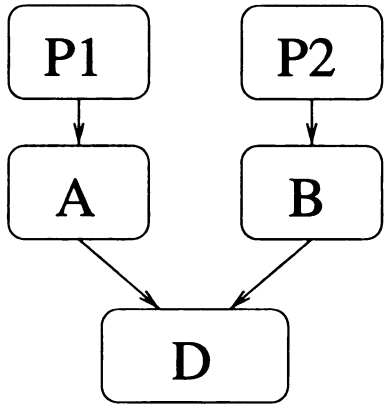

Figure 7: A Composability Problem

and interleaved with the computation. Observe that the encapsulation of the mechanism in a monitor does not serve to localize synchronization information. Instead, the monitor functions as a central authority to avoid deadlock problems, but contains code that belongs in the client, thereby violating information hiding. One indication of this violation is that adding more clients (philosophers) requires the modification of the monitor code.

We recognize this problem as one of poor separation of concerns; the migration of client code to a central monitor is due to the exigency of preventing deadlocks, making it impossible to factor the code out. A side effect of this poor separation of concerns is a lack of reliability. Code such as the one in Figure 6 is hard to verify, and the effect of changes is not always apparent. Moreover, there is no mechanism in the code in Figure 6 to indicate whether we have proper mutual exclusion.

Another problem encountered here is that of composability. If we were not using a *central* locking mechanism in the code, adding more clients would run the risk of correctness problems, most importantly race conditions and deadlocks. There are solutions that avoid the need of a central locking mechanism (as discussed in Section 2.3.2), but they are diffi-

cult to use correctly in an ad-hoc fashion. We cannot just connect modules together as we do for sequential code and have some confidence that the combination works as expected.

This problem of composability becomes particularly visible if one considers a hierarchically layered system. How does one specify process interaction in such a way that hidden implementation details at lower layers are not exposed to higher layers? For instance, suppose there are two processes $P1$ and $P2$, accessing a shared object D through separate front ends A and B (see Figure 7).

Unfortunately, making D visible violates the principle of information hiding—the processes should only access the objects A and B directly, and their dependence on D should not be exposed to the implementation of $P1$ or $P2$. But if D isn't visible to $P1$ or $P2$, multi-step transactions by $P1$ and $P2$ are likely to interfere with one another, because $P1$ and $P2$ do not know that they have to lock D for the duration of such a transaction. Moving the requirement to lock D to the code of A or B would lead to the same delocalization as observed in the previous example.[2]

To solve this problem, one needs to determine the set of shared objects that are affected by a transaction. This cannot be done in an ad-hoc fashion by the programmer without exposing information that is private to A and B. The best way to infer the set of affected shared objects would be to have the compiler or the runtime system infer the set of affected objects automatically or semi-automatically. To allow the compiler or runtime system to perform such inference requires in general that synchronization dependencies be declared, i.e. a contractual approach be used.

[2]Note that while certain existing language mechanisms, such as Sather's multi-branch locks or SCOOP, can handle transactions that access a *known* set of shared objects, they do not effectively handle layered systems.

2.6 A Classification of Synchronization Mechanisms

We now turn to the discussion of non-trivial synchronization mechanisms for passive objects. We call a synchronization mechanism non-trivial if it can handle the atomic acquisition of multiple shared resources. Existing approaches specify the resources that are to be acquired in a variety of ways. We organize these approaches according to a hierarchy of synchronization contracts and classify the power of synchronization mechanisms according to this hierarchy.

Specifically, we say that a synchronization mechanism is *obviated by* a family of synchronization contracts, if the behavior implemented by the mechanism can be specified declaratively using such contracts. The family of synchronization contracts that obviates a synchronization mechanism therefore provide an upper bound on the expressiveness of the mechanism without regards to implementation details. We adopt the following notational conventions to describe synchronization contracts. An exclusion contract between a client C and a supplier S is denoted $C \rightarrow S$, and a conditional contract between a client C and a supplier S that depends on a predicate P is denoted $C \xrightarrow{P} S$. Also, if A and A' are synchronization contracts, then $A \wedge A'$ and $A \vee A'$ describe their explicit conjunction and disjunction, respectively. We do not explicitly consider parameterized contracts, because the predicate on which a parameterized contract depends is not itself a contract but rather an enabling condition on the actual contract.

We begin the classification by exploring two common approaches to synchronization—two-phase locking and general resource allocation—and show that these approaches are obviated by the explicit composition of exclusion contracts (Section 2.6.1). Such approaches

provide the bottom-most tier in our hierarchy of synchronization mechanisms, although the underlying deadlock avoidance and recovery mechanisms are the foundation on which more powerful mechanisms have been built. We conclude with the survey of recent research into more powerful synchronization mechanisms, specifically those that provide for the explicit composition of conditional contracts, and explicit composition of contracts by disjunction.

2.6.1 Existing Approaches for Explicit Composition of Synchronization Contracts

The most common of the non-trivial synchronization mechanisms are obviated by the explicit composition of exclusion contracts. The client in such a contract is an active entity, such as a process or a job running within a process. Clients employ a synchronization mechanism (or collaborate with other clients according to a synchronization protocol) to gain exclusive access to one or more suppliers. Each supplier is a passive entity, such as a database record, a shared operating system resource, or in-memory data accessed as part of a session in an e-commerce application. We discuss two typical styles of such mechanisms, namely two-phase locking and resource allocation.

Additionally, there are also synchronization mechanisms that are more powerful in the sense that they cannot be obviated by the explicit composition of exclusion contracts. Instead, they can only be obviated by the explicit composition of exclusion *and* conditional contracts. Likewise, there exist other contractual synchronization models besides the universe model that allow for the explicit composition of conditional contracts. We discuss

both of these types of contracts in Section 2.6.2.

Two-phase Locking Schemes

Two-phase locking [EGLT76] is a mechanism for consistent concurrent access to shared resources in transaction-based systems, such as databases [BK91], CORBA Concurrency Control Services [SKT+96], and multi-user collaboration systems for software development or CAD/CAM [ABAK95, Kai95]. In the two-phase locking model, a transaction consists of two phases. In the first or *growing phase*, all shared resources are acquired, while in the second or *shrinking phase*, all shared resources that have been acquired during the growing phase are released. Resources can be acquired either *en bloc* at the start of the growing phase, or on demand, as they are needed by the transaction, at any time during the growing phase. They are, however, never released before the beginning of the shrinking phase, and no further resources can be acquired once the shrinking phase has begun.

During the growing phase, *concurrency control mechanisms* are employed to ensure that no deadlock occurs. Examples of concurrency control mechanisms for two-phase locking are *wound-wait* [RSL78], *wait-die* [RSL78], *cautious waiting* [HZ92], *running priority* [FR85], *immediate restart* [SPG91], *general waiting* [BHG87] and *wait-depth limited (WDL)* [FHRT92]. Of these, only *wound-wait* is automatically deadlock- and starvation-free. *Wait-die*, *cautious waiting*, *running priority*, and *immediate restart* are deadlock-free, but not starvation-free, and must be augmented with a mechanism to avoid starvation. *WDL* and *general waiting* are not necessarily deadlock-free. To avoid deadlock using one of these schemes requires a background process [ACM87, Che95] that determines if there is a cycle of transactions waiting for one another. Because each of these algorithms vies for a fixed

set $\{R_1, \ldots, R_k\}$ of resources, they are collectively obviated by synchronization contracts of the form $T \rightarrow R_1 \wedge \ldots \wedge T \rightarrow R_k$, where the client is a transaction T.

Most approaches to two-phase locking attempt to hide the details of concurrency control from the programmer in ways that do not involve synchronization mechanisms or contracts. In particular, many transaction-based systems do not require transaction code to explicitly acquire resources before using them, thus making resource acquisition transparent to the programmer. For example, the Enterprise JavaBeans specification suggests that a resource should be automatically locked the first time it is accessed [DYK01] without being directed to do so explicitly by the programmer. Such automatic locking can eliminate the need for both explicit synchronization mechanisms and explicitly specified synchronization contracts.

Unfortunately, these automatic locking mechanisms recover from deadlock by rolling back transactions. While transaction-based systems in general support rollback, it is practically infeasible to integrate rollbacks into a general purpose programming language. Rolling back an arbitrary computation would involve storing and later recovering the entire state of a process, including processor registers, the processor stack, and the state of the operating system kernel with respect to the process, which in itself can be difficult or impossible [EAWJ96]. In addition, certain operations, such as I/O or the creation of processes, cannot be undone at all, and have to be delayed until no rollbacks are possible. This is a common requirement of rollback-based systems such as [Jef85], and requires pervasive changes to the language implementation and libraries. Finally, the need to store duplicate state information requires CPU and memory overhead, which can be considerable [CPF99].

In summary, synchronization contracts obviate a large class of synchronization mech-

anisms, as they are commonly used in transaction-based systems. Unlike mechanisms that require rollback capability, a contractual approach can be supported by general purpose programming languages.

Resource Allocation in Distributed Systems

Synchronization mechanisms for composite synchronization contracts have also been investigated extensively for distributed message passing systems, generally under the term *resource allocation*. Resource allocation assumes that there is a set of processes, such that each process P has a corresponding set R of resource requirements. Most commonly, each R is simply a set $\{R_1, \ldots, R_k\}$ of resources, all of which must be exclusively acquired by the corresponding process P. Deadlock-free acquisition protocols for such requirements can be found in [CM84, Rhe98, CS99]. This type of resource allocation protocol is obviated by synchronization contracts of the form $P \rightarrow R_1 \wedge \ldots \wedge P \rightarrow R_k$. Alternatively, each R can also be a set of sets $\{\{R_{1,1}, \ldots, R_{1,k_1}\}, \ldots, \{R_{l,1}, \ldots, R_{l,k_l}\}\}$ of resources, where exclusive access to one of the $\{R_{i,1}, \ldots, R_{i,k_i}\}$ is needed to satisfy the resource requirements of P. Examples of deadlock-free acquisition protocols for these requirements can be found in [BB98, BBF01]. They are obviated by synchronization contracts of the form $(P \rightarrow R_{1,1} \wedge \ldots \wedge P \rightarrow R_{1,k_1}) \vee \ldots \vee (P \rightarrow R_{l,1} \wedge \ldots \wedge P \rightarrow R_{l,k_l})$.

2.6.2 Beyond Composition of Exclusion Contracts

In addition to the common mechanisms discussed so far, a number of more powerful synchronization mechanisms have also been proposed. These more powerful mechanisms are not obviated by the arbitrary composition of exclusion contracts, but are obviated by the

composition of exclusion and conditional contracts. We present two representative synchronization mechanisms [FP97, JP93] to illustrate this point. We then describe two contractual synchronization models [Mey97, Hol99] that support the composition of conditional contracts.

One of the more powerful procedural synchronization mechanisms that has been implemented in a programming language is Sather's multi-branch locking statement [FP97]. Its basic syntax is:

lock

when $L_{1,1}, \ldots, L_{1,k_1}$ **then** *statement_list*$_1$

...

when $L_{l,1}, \ldots, L_{l,k_l}$ **then** *statement_list*$_l$

end

When executed by a process P, this statement is used to regulate exclusive or conditional access to a set of shared resources. Each $L_{i,j}$ can be either a mutex or a semantic condition. A mutex provides exclusive access to an object $S_{i,j}$, whereas a semantic condition provides exclusive access to an object $S_{i,j}$ once $L_{i,j}$ has been acquired (if it is a mutex) or is satisfied (if it is a semantic condition).[3] As soon as all of the locks and conditions of a given **when** branch can be locked or verified, respectively, the corresponding *statement_list*$_i$ is executed and has exclusive access to all $S_{i,j}$. Let $A_{i,j}$ be the exclusion contract $P \rightarrow S_{i,j}$ if $L_{i,j}$ is a mutex, or the conditional contract $P \overset{L_{i,j}}{\rightarrow} S_{i,j}$ if $L_{i,j}$ is a semantic condition. Then each **when**

[3]When we say that the mutex or condition provides exclusive access to an object, we describe the intended use of this construct. This use is not enforced by the language implementation. For example, Sather does not actually associate mutexes with specific shared resources, and it is legal to access a shared object without executing a lock statement at all.

branch of the multi-branch lock is obviated by the conjunction of those $A_{i,j}$ that obviate the $L_{i,j}$ of that particular branch. The entire multi-lock statement is obviated by a disjunction of these conjunctions, namely $(A_{1,1} \wedge \ldots \wedge A_{1,k_1}) \vee \ldots \vee (A_{l,1} \wedge \ldots \wedge A_{l,k_l})$.

In addition to the basic syntax above, Sather also allows the statement to have an else-branch that is taken when none of the when-branches can be taken. The behavior of such an else-branch cannot be obviated by synchronization contracts that conform to our existing classification.[4]

Another powerful synchronization mechanism is expressed by the **holdif** statement of the Concurrent Extensions of Eiffel [JP93]. The syntax of the **holdif** statement is **holdif** *condition* **then** *statement_list* **end**. The *condition* of the **holdif** statement can be an arbitrary semantic predicate involving one or more supplier objects, $S_1, \ldots, S_n$. The semantics prescribe that all the S_i are locked atomically as soon as the *condition* is true. Once that occurs, the *statement_list* will be executed. Consequently, the **holdif** statement is obviated by composite conditional contracts of the form $P \overset{Cond}{\rightarrow} S_1 \wedge \ldots \wedge P \overset{Cond}{\rightarrow} S_n$, where *Cond* is the *condition* of the **holdif** statement and P is the process executing it. Note that *Cond* can be a predicate that depends on the state of more than one of the S_i, such as $S_1.size + S_2.size \geq M$, with M being a constant.

A contractual approach that obviates the CEE **holdif** construct is described in the SCOOP extension of Eiffel [Mey97]. SCOOP expresses synchronization contracts implicitly through the type and contract signature of an Eiffel routine. An Eiffel routine may have arguments that are shared suppliers $S_1, \ldots, S_n$, and, optionally a precondition *Pre*. Such

[4]However, in Section 10.1.1 we will discuss how to extend that classification to cover alternative interpretations of disjunctions that would obviate the behavior of the else-branch.

a routine functions as a synchronization contract insofar as entry into the routine body is blocked until exclusive access to all of the S_i has been obtained and until *Pre* is verified. This expresses a synchronization contract of the form $P \rightarrow S_1 \wedge \ldots \wedge P \rightarrow S_n$ if the routine has no precondition, and $P \overset{Pre}{\rightarrow} S_1 \wedge \ldots \wedge P \overset{Pre}{\rightarrow} S_n$ if there is a precondition *Pre*. In either case, P is the process calling the routine.

The synchronization rings proposed in [Hol99] are another contractual approach. Conceptually, a synchronization ring is a wrapper around one or more shared supplier objects that intercepts access to these suppliers, permitting access only when one or more contracts (called *constraints*) are satisfied, and blocking access otherwise. More than one synchronization ring can be wrapped around a supplier, and in this case all the constraints provided by any of the rings must be satisfied for a process to be granted access to the supplier wrapped by these rings. The basic contracts that can be expressed by synchronization rings are exclusion and conditional contracts, and by wrapping multiple synchronization rings around the same supplier, they can effectively be composed by conjunction.[5] An enhancement that synchronization rings provide over synchronization contracts as expressed in our classification scheme is that they allow for finer granularity of exclusion or conditional contracts, and can specify that two processes can access a shared supplier concurrently, if they do not interfere with one another within that supplier.[6]

[5]The synchronization ring model also offers shorthand forms of commonly used conditional contracts as well as contracts that ensure quality of service, but these contract types are beyond the scope of this thesis.

[6]Concepts that can express such concurrent access to a shared supplier are beyond the scope of the universe model, but will be addressed in detail in Section 10.1.2.

Chapter 3

The Universe Model

The universe model supports the declarative specification and run-time negotiation of exclusion and conditional synchronization contracts, parameterized contracts, and their composites. Under this model, processes operate in conceptually disjoint data spaces, which expand and contract over the lifetime of a program in order to satisfy the synchronization contracts of each process. We first introduce these ideas informally, in the context of an example (Section 3.1), followed by a more formal description of the basic concepts (Section 3.2). Contracts take the form of data invariants, called *concurrency constraints*, and are associated with modules that encapsulate cohesive groups of objects, called *universes* (Section 3.3). A run-time system negotiates these contracts by *migrating* universes among the data spaces of the competing processes in order to satisfy the contracts that pertain to these universes. The run-time system also schedules or suspends processes to guarantee that dataspaces contain the necessary universes while ensuring the dataspaces remain disjoint (Section 3.4). A formal semantics of negotiation appears in [SBD03]. In the sequel, we refine the notion of a data space into a connected, directed graph of universes in shared

memory, thus enabling migration among data spaces to be implemented by linking and unlinking edges. Moreover, we refine the functionality of the run-time system into a procedure that is invoked by each process to update its graph in a way that satisfies contractual obligations.

3.1 Motivating Example

Consider the design of a web-based whiteboard, generally known as a *wiki* [LC01], which is used for on-line collaboration. A wiki allows a group of authors to use a web browser to concurrently view, create, and edit text pages that are maintained in a shared repository. Each page contains marked-up text with simple formatting instructions and hyperlinks to other pages. Figure 8 depicts the design of classes to represent the pages that can be stored and manipulated in a wiki.[1] Pages can be either permanently stored on secondary storage (`REAL_PAGE`) or created dynamically from other data (`VIRTUAL_PAGE`). Virtual pages generally provide various types of information about the system, such as lists of changes, differences between two versions of a page, the local or global hyperlink structure, unreachable pages, or resource usage. To avoid accidental deletion of data, updating a real page does not simply overwrite the old version, but stores that old version in a database, so that it can be restored at a later date. In our design, the current version of a real page is always kept resident in memory, which allows for rapid browsing of pages, including global full-text searches. Of course, keeping these pages in memory requires careful concurrency

[1] The diagram uses the UML class diagram notation [BJR99], where boxes denote classes, lines with a triangle denote generalization relationships where the triangle points to the more general class, and arrows denote references.

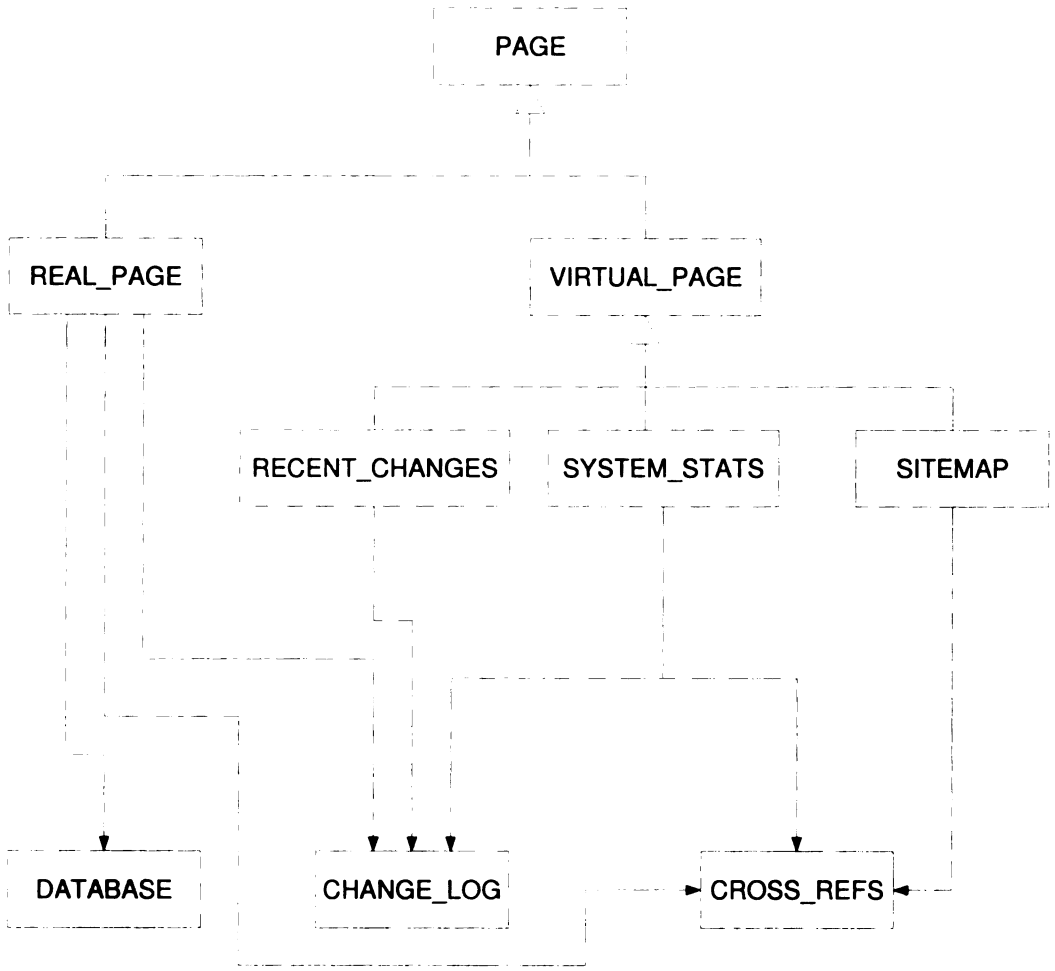

Figure 8: UML model of page classes in the wiki example.

management to avoid data races, starvation, and deadlock.

Figure 9 depicts part of the definition of class `REAL_PAGE` in our extended version of Eiffel. The class definition is prefixed by the keyword **universe** (line 1), which indicates that instances of this class may participate in the negotiation of synchronization contracts. For brevity, we defer the formal definition of universe classes to Section 3.2. Lines 3–9 declare class attributes of which there are three different types. The attributes `query_db` and `updating` are called *condition variables*; they encode different synchronization states of instances of this class and may be referenced in the premise of a parameterized con-

```
1   universe class REAL_PAGE inherit PAGE
2   feature -- attributes
3       id: STRING -- name of this page
4       current_source: STRING -- most recent version of this page
5       history: LIST[VERSION] -- list of valid versions
6       database: DATABASE
7       change_log: CHANGE_LOG
8       cross_refs: CROSS_REFS
9       query_db, updating: BOOLEAN
10  feature -- methods
11      create_new_version(source: STRING) is
12      local now: TIME; version: VERSION
13      do
14          updating := true
15          now := get_current_time
16          version := database.add_new_version(id, source, now)
17          cross_refs.delete_links(id, links(current_source))
18          current_source := source
19          cross_refs.add_links(id, links(current_source))
20          change_log.register_change("New Version", id, version, now)
21          history.prepend(version)
22          updating := false
23      end
24  concurrency
25      (query_db or updating) => database
26      updating => change_log and cross_refs
27  end -- class REAL_PAGE
```

Figure 9: Elided definition of class REAL_PAGE in (universe-model extended) Eiffel.

tract or the condition of a conditional contract. The attributes `database`, `change_log`, and `cross_refs` are called *universe variables*; they reference objects that are instances of universe classes.[2] Finally, attributes `id`, `current_source`, and `history` are local object references that are neither condition nor universe variables.

A universe is a cohesive group of objects that migrate together during contract negotiation. By virtue of being declared a universe class, instantiating REAL_PAGE creates not

[2]for brevity, the declarations of these other classes are elided in Figure 9

only an instance of the class but also a new universe in which the instance is placed. This will contain the actual instance of class REAL_PAGE, hereafter called the *root object* of the universe. Here, and in the sequel, when we refer to a universe in a context that requires an object, we mean the root object of the universe. The universe will also contain any objects referenced by any (non-universe variable) attributes of an object that is already in the universe. Thus, for example, the universe will also contain the objects referenced by attributes `id`, `current_source`, and `history`, along with any object references in these objects recursively.

Lines 11–23 define a method that creates a new version of a page. This code adds the new version to the global database, registers the change in the change log, and updates the cross reference information. Synchronization is governed by the contract, or *concurrency constraint*, which specifies that instances of the class may access `database` if `query_db` or `updating` is true and that the class may also access `change_log` and `cross_refs` if `updating` is true (lines 25–26). Only condition and universe variables may be referenced in a synchronization contract.

Observe that `create_new_version` does not use low-level synchronization mechanisms to guarantee mutual exclusion. Instead, it simply manipulates the boolean attribute `updating`. The run-time system uses the values of `updating` and `query_db`[3] to determine which objects need to be migrated into the data space of the current process, and this migration is performed automatically, without further intervention by the programmer. Conceptually, boolean variables such as `query_db` and `updating` en-

[3]The `query_db` attribute is not manipulated in this example.

code the abstract state of a universe; whereas universe references, such as `database`, `change_log`, and `cross_refs`, denote shared resources that are needed by instances of `REAL_PAGE` to implement its methods. The contract relates different configurations of the abstract state to different resource requirements. Specifically, each universe reference—`database`, `change_log`, and `cross_refs`—denotes an exclusion contract with the referenced universe; the predicates `updating` and `query_db or updating` denote different synchronization-relevant state conditions; and the combination of these entities by implication and conjunction constitutes a parameterized composite contract.

3.2 Formal introduction to the universe model

A program in the universe model comprises one or more sequential *processes*, each of which is a thread of control whose data space, or *realm*, comprises a collection of universes that the process is said to *own*. A universe (and thus any object contained within a universe) cannot be owned by more than one process concurrently. In addition, a process cannot legally access an object in a universe that it does not own. For example, accessing the object referenced by the universe variable `database` (line 16 of Figure 9) would be illegal if this object was not in the accessing process's realm. Our model prescribes that an illegal access results in a run-time exception. Consequently, programs written using the universe model are guaranteed to be free of so-called *data races* as defined in [NM92].

Processes communicate indirectly using a mechanism called *universe migration*, through which the owner of a universe changes at run time. Over the lifetime of a program, a universe might change owners many times. Indeed, there could be times when a

universe is not owned by any process at all. Process scheduling is managed by a run-time system that attempts to migrate universes so as to satisfy the requirements of synchronization contracts. A process can be scheduled only if every contract that pertains to every universe in its realm is satisfied; we call such a realm *sufficient.*

A programmer manipulates a universe by performing operations on a distinguished group member called the *root object.* Each universe has exactly one root object, and the creation/destruction of a universe coincides with the creation/destruction of its root. A root object is distinguished as such by being an instance of a *universe class*, which (in Eiffel) we denote using the keyword **universe**, as depicted in Figure 9 (line 1). Attributes of a universe class may be condition variables, universe variables, or regular attributes that are not used to declare a contract. Instantiating a universe class creates both a new object and a new universe in which to place the newly instantiated object. Instantiating a non-universe class creates a new object and places it in the same universe as the object that invoked the creation procedure to construct the newly instantiated object.

During the execution of a program, a process will occasionally execute an instruction that affects the contents of its realm. In Figure 9, for example, the assignment to the instance variable `updating` on line 13 affects the realm because the contract requires `database`, `change_log`, and `cross_refs` to be a part of the realm whenever `updating` is true. Likewise, the assignment on line 22 affects the realm. When a process invokes one of these *realm-affecting operations*, execution of the process is suspended and the realm is adjusted by universe migration. This adjustment involves both removing unneeded universes from the realm and adding universes that are required to make the realm sufficient. If the adjustment is successful, then the process can resume execution; otherwise, the process is

```
1    universe class EVENT_MONITOR
         ...
2    feature { NONE } – private attributes
3        channel : CHANNEL – universe variable
4        reading : BOOLEAN – condition variable
         ...
5    concurrency
6        reading => channel when channel.has_data
7    end – class EVENT_MONITOR
```

Figure 10: Elided definition of class EVENT_MONITOR. Notice constraint includes a conditional universe reference.

blocked until its needs can be satisfied.

3.3 Contract language: Concurrency constraints

Contracts in our model are expressed as *concurrency constraints*, which specify the conditions under which operations over objects in a given universe might access objects in other universes. A concurrency constraint is a limited propositional formula, whose atomic propositions are either conditions or universe references.

3.3.1 Conditions and Condition Verification

A *condition* is a (full) propositional formula, whose atomic propositions are condition variables. We distinguish two classes of conditions—local and remote. The syntax of a *local condition* is defined as follows:

$$\begin{aligned} \textit{local\_condition} \;::=\; & \textit{true} \\ \mid\; & \textit{conditionVar} \end{aligned}$$

| **not** *local_condition*

| *local_condition* **and** *local_condition*

| *local_condition* **lor** *local_condition*

| *local_condition* $\Rightarrow$ *local_condition*

Here, the terminal symbol *conditionVar* refers to a condition variable, and the terminal symbol *true* is the trivial condition. Also, the usual precedences and associativities of logical operators apply.

Likewise, a *remote condition* is a full propositional formula, whose atomic propositions are either condition variables or a *remote condition variables*, which are condition variables of a universe that is referenced indirectly through a universe variable. More formally:

remote_condition ::= *true*

| *conditionVar*

| *universeVar* . *conditionVar*

| **not** *remote_condition*

| *remote_condition* **and** *remote_condition*

| *remote_condition* **lor** *remote_condition*

| *remote_condition* $\Rightarrow$ *remote_condition*

Here, the terminal symbol *universeVar* refers to a universe variable, and (while not shown in this grammar) a remote condition may reference at most one universe variable. For

example, $u.x$ **and** $u.y$ is a syntactically legal remote condition; whereas $u.x$ **and** $v.y$ is illegal because it references two universe variables (u and v). The atomic constraints of the concurrency constraint on line 6 in Figure 10 illustrate both types of conditions, where `reading` and `channel.has_data` denote local and remote conditions, respectively.

A condition is verified by universes, the root objects of which provide value assignments to condition and universe variables that appear in the condition. Let u be a universe and c be a local condition whose free variables refer to condition variables in u. We say that u *verifies* c if, by substituting the value $u.x$ for each condition variable x that is free in c, the resulting formula is logically valid. Remote conditions must be verified by two universes, a local universe (u) and a remote universe (v). Let c be a remote condition all of whose free variables refer either to a condition variable in u, a condition variable in v, or a distinguished universe variable x in u. We say that u and v verify c if, by substituting the value $u.y$ for each local condition variable y in c, and by substituting $v.y$ for each remote condition variable $x.y$ in c, the resulting formula is logically valid.

3.3.2 Constraints and Constraint Satisfaction

Syntactically, concurrency constraints are limited propositional formulae whose atomic propositions are either conditions or *universe references*, which specify mutually-exclusive access dependencies. Here, and in the sequel, we adopt a typographic distinction between conditions and constraints, using using lower-case italic letters at the beginning of the alphabet (e.g., c, c') to refer to conditions and capital calligraphic letters at the beginning of the alphabet (e.g., $\mathcal{C}$, $\mathcal{C}'$) to refer to constraints. The formal syntax of a concurrency

constraint is as follows:

constraint ::= *true*

| *universeVar*

| *universeVar* **when** *remote_condition*

| *local_condition* ⇒ *constraint*

| *constraint* **and** *constraint*

| *constraint* **or** *constraint*

A constraint may be the trivial constant true (line 1), a definite universe reference (line 2), a conditional universe reference (line 3), a contextual constraint (line 4), or a composite constraint (lines 5-6). A definite universe reference asserts that the referent is in the same realm as the client universe to which the constraint applies. The reference to `change_log` in line 26 of Figure 9 is an example of a definite universe reference.

A conditional universe reference also asserts that the referent is in the realm, but it contains an additional condition (called a *when clause*) that is used to *defer* the migration of a universe into a realm until the condition can be verified. For example, the expression `channel when channel.has_data` (line 6 of Figure 10) is a conditional universe reference.[4] A contextual constraint of the form $c \Rightarrow C$ asserts C whenever condition c holds. Definite and conditional universe references are used to declare exclusion and con-

[4]This example is described in detail in [SBD03]. Also, notice that the when-clause is a new feature added after the publication of [BS00]. It replaces: *reading* ⇒ *queue.has_data* ∧ *queue* with *reading* ⇒ *queue* **when** *queue.has_data*.

ditional contracts respectively;[5] whereas contextual constraints are used to declare parameterized contracts.

Semantically, we say a concurrency constraint is satisfied by a universe in the context of a set of dependent universes. Here, and in the sequel, we carefully distinguish the term *verify*, which we use to evaluate conditions, with *satisfy* (or *locally satisfy*), which we use to evaluate constraints. Let U be a set of (root objects of) universes, $u \in U$ be a specific object in this set, and C be a well-typed constraint that pertains to u. We say that C is *locally satisfied* by u in the context of U if C is logically valid under the following substitution:

1. Each local condition c that appears as the premise of a contextual constraint in C is replaced with the value:
   - true if u verifies c; and
   - false otherwise
2. Each definite (resp. conditional) universe reference v (resp. v **when** c) in C is replaced with the value:
   - true if $u.v$ is either `null` or an element of the set U.
   - false otherwise

Observe that there are two distinct cases to consider in the local evaluation of a universe reference. First, the referent could be `null`, in which case we interpret the reference as vacuously true.[6] On the other hand, the referent might not be `null`, in which case we

[5] A conditional contract $C \xrightarrow{P} S$ in the universe model requires that P depends only on the state of C and S, but on no other universes.

[6] See [SBD03] for the rationale behind this decision.

interpret the reference as true provided that the universe u' associated with the referent is contained in the context U. Observe also that an unverified when clause can only defer migration into a realm; it does not affect the local satisfiability of a constraint and thus cannot cause the referent universe to be migrated out of a realm.

For example, the full constraint in Figure 10 enforces the following behavior: While `reading` is false, nothing is required; however, once a process assigns `reading` the value true, if the referent of `channel` is non-null, then the process may not continue unless the referent is in its realm. If the referent is not already in the process's realm, its migration into the realm will be deferred until such time as the attribute `channel.has_data` is set (by another process) to true. This design not only ensures mutual exclusion; it also causes `EVENT_MONITOR` objects to block until the channel contains data. Moreover, once the channel is in the realm, it will remain there until the process sets `reading` to false, even if some operation on the channel causes `channel.has_data` to become false, which could happen for example if the process empties the channel. Thus, a when clause is merely a mechanism for deferring migration into a realm; it cannot be thought of as an invariant that holds whenever the referent universe is in the realm.

We use local satisfaction to formally define the sufficiency of a realm. A set U of universes is sufficient if, for every $u \in U$, we can show that u locally satisfies its associated concurrency constraint in the context of U. Sufficiency is the basis for process scheduling and universe migration, which we now discuss.

3.4 Contract Negotiation and the Run-time System

The universe model includes the concept of a run-time system, which is responsible for negotiating contracts and scheduling processes for execution. Under this model, the operational state of a process alternates between being *runnable* and being *blocked*, and a runnable process remains runnable until it executes a realm-affecting operation, i.e., it modifies a condition or universe variable in some universe in its realm. A programmer using the universe model understands the synchronization effects of her program by thinking of each such operation as *trapping* into the kernel of this run-time system. The idea is that a realm-affecting operation could create a new universe or modify the parameters of a parameterized or conditional contract, all of which could cause active contracts to be re-negotiated. In response to such an operation, the run-time system then renegotiates any affected contracts and returns control to a process once all of its relevant contracts are satisfied.

More precisely, a process is deemed runnable if its realm satisfies the following conditions. At minimum, the realm of a process must contain any universe whose root object is actively executing a method. That is, the realm must contain any universe whose root is the object of a method[7] with an activation record on the call stack. We refer to the set of such universes as the *universe stack* and denote it using the symbol S_p where p is the process to which it applies. In addition to these minimum requirements, the realm of a runnable process must be sufficient and minimal. More formally, a blocked process p with realm U

[7] By "object of a method", we mean the object referred to by the keyword `this` in C++, `self` in SmallTalk, or `Current` in Eiffel. An activation record is created with a `this` pointer that references object `ob` whenever a statement of the form `ob.method(...)` is invoked.

is deemed runnable if and only if:

$$
\begin{array}{l}
S_p \subseteq U \\
\quad \wedge \\
\mathit{sufficient}(U) \\
\quad \wedge \\
\neg \exists\, U' \subset U \mid S_p \subseteq U' \ \wedge \ \mathit{sufficient}(U')
\end{array}
$$

We refer to a realm that satisfies these conditions as a *complete realm*, and we refer to the process of making a blocked process runnable as *updating* the realm.

The run-time system is responsible for updating the realms of all the processes in the system. This is accomplished by migrating universes among the realms of all contending processes in order to complete as many realms as possible. Thus, the operational semantics of process scheduling are defined entirely in terms of universe migration.[8] A major contribution of this thesis is a distributed algorithm for negotiating universe-model contracts by simulating the migration of universes among the realms of competing processes. Because real threads operate over shared memory as opposed to private data spaces, universe migration need not involve physically copying the objects to new memory locations. Rather, each universe is attributed by a handle that identifies the owner process, and migration is simulated by manipulating that attribute. More generally, the realm-update procedure is run independently by each process in a running system and several processes may be running the realm-update procedure for their respective realms concurrently; universe migration is implemented as a protocol of universe acquisition and release, such that processes will fairly negotiate for contended universes and a process will block if it cannot acquire all of

[8]For a detailed treatment of these semantics see [SBD03].

the universes it needs; and acquisition and release are implemented by manipulating data in shared memory. The challenge to implementing such an algorithm is to efficiently represent realms and to design the algorithm so that processes running the algorithm avoid starvation and deadlock.

Chapter 4

Realm update

Our realm-update procedure, which is invoked when a process performs a realm-affecting operation, renegotiates the contracts of the invoking process and returns once this renegotiation concludes and the realm of the process is once again complete. Before delving into the algorithms used in this procedure, we first show how a realm is a directed graph of universes that satisfy a reachability property and that we can represent the realm to which a given universe belongs using a marking strategy (Section 4.1). Our algorithms maintain the reachability property using depth-first and breadth-first search procedures that consult and update these markings. The algorithm that releases unneeded universes is based on concepts from garbage collection [Wil92], specifically the use of reference-counting collection with cycle detection [BAR+03] (Section 4.3). By contrast, the algorithm that acquires new universes (Section 4.4) combines several ideas to fairly migrate universes while avoiding or automatically recovering from a large class of deadlock conditions (Chapter 5).

4.1 Run-time representation of realms

To motivate our implementation of realms and the algorithms that maintain them, we begin by viewing the set of universes in a running system as a directed graph. Let $\mathcal{G}$ represent the *global universe graph*, whose nodes correspond to the universes in a running system, and each of whose directed edges corresponds to a link from a source to a referent universe through a universe variable. Then a complete realm R_p of a process p is a subgraph of $\mathcal{G}$ that contains the universe stack S_p and all nodes reachable via a path of so-called *accessible* edges. An edge $u \rightarrow v \in \mathcal{G}$ is accessible if v is in a minimal set of universes that are needed to locally satisfy u's constraint. In such a case, we also say that v is accessible from u. For instance, if u has the constraint $(c \Rightarrow v_1) \wedge (\neg c \Rightarrow v_2)$ and c is true, then the edge $u \rightarrow v_1$ is accessible, whereas $u \rightarrow v_2$ is not accessible.

The function `accessibles` returns a set of target nodes that are accessible from a given source node:

function accessibles(u : Universe) **returns** Set[Universe];

This function is computed based on the concurrency constraint associated with `u` and the values of condition variables in `u`. For efficiency, our compiler generates a custom `accessibles` function for each universe class using a symbolic encoding based on binary-decision diagrams (BDDs) [Bry86]. Note that `accessibles(`u`)` is only well-defined when `u` is in the realm of the current process.

Because concurrency constraints may contain disjunctions, the set of accessible targets is non-deterministic. The implementation of `accessibles` resolves the non-determinism by choosing from among the multiple possible choices and remembering the choice. More

precisely, the choice made by `accessibles(`u`)` persists until u witnesses a realm-affecting operation or is migrated to another realm.

The extent of a realm can be computed using reachability algorithms, such as breadth-first search, by recursively exploring the accessible edges that emanate from nodes on the universe stack. In the sequel, we present algorithms that update a realm dynamically, following a realm-affecting operation. To support these algorithms, each universe contains an *owner* attribute, which records the process that currently owns the universe, and a *reference count*, which records the number of universes in the same realm from which this universe is accessible. More precisely, the reference count of a universe u, with respect to a process p, is the number of accessible edges $v \rightarrow u$ where u is the target universe, plus one if $u \in S_p$. These attributes are accessed using the functions `owner` and `RC` respectively. The function `raccessibles(`u`)` (short for "realm" accessibles) returns the set of universes that are in the set `accessibles(`u`)` and also in the same realm as u (i.e., for which the owner is the same is the owner of u).

To implement the algorithms that follow, we rely on a few basic synchronization primitives. First, we assume that every universe u and every process p has an associated mutex, which can be locked by the instructions `LOCK(`u`)` or `LOCK(`p`)`, respectively, and similarly be unlocked by the instructions `UNLOCK(`u`)` or `UNLOCK(`p`)`. Second, to handle waiting for conditions, we assume the presence of an abstract **wait until** primitive with the following syntax and semantics.

wait until $cond_1$ **then** $statement_1$

or until $cond_2$ **then** $statement_2$

...

or until $cond_n$ **then** $statement_n$

end

A **wait until** statement suspends the current process until one of the conditions $cond_i$ is true, at which time the corresponding $statement_i$ is executed. We assume that when a process p executes a **wait until** statement, checking for conditions is an atomic operation enclosed in `LOCK(`p`)`...`UNLOCK(`p`)` statements. If more than one $cond_i$ is true, the first alternative in the list is chosen. We have designed the $cond_i$ so that a straightforward semaphore-based implementation is possible.

4.2 Realm-update procedure

A realm-affecting operation causes an implicit release of a set R^- of universes that are no longer required and an implicit request for a set R^+ of new universes needed to complete the new realm. The semantics of the universe model guarantee that every universe in $R^- \cup R^+$ is reachable from the universe w that *witnessed* the realm-affecting operation. Our realm-update procedure computes and releases R^- using a procedure called *realm contraction* and then computes and acquires R^+ using a procedure called *realm completion*.

Both R^+ and R^- can be derived from the change to `accessibles(`w`)` and the change to the universe stack S_p between the previous realm update and this one. The change to `accessibles(`w`)` could make edges inaccessible that were previously accessible from w or vice versa. Likewise, the change in S_p could make inaccessible those universes that are no longer on the stack, or make accessible those universes that have been

added to the stack since the previous realm update. Thus, to update a realm requires knowing: (1) the value of `accessibles(`w`)` immediately before and immediately after the operation, and (2) the contents of the universe stack immediately following and immediately preceding the previous and current realm-update operations respectively.

Let `old_ustack(p)` be the universes in S_p immediately following the previous realm update, and let `ustack(p)` be the set of universes in S_p upon invocation of the current update procedure. Let `old_acc` be the set of universes accessible from w after the previous realm update and `new_acc` be the set of universes accessible from w at the beginning of this update (i.e., immediately following the most recent realm-affecting operation). R^- is then the set of universes that become inaccessible as a result of deleting edges from w to universes that occur in `old_ustack(p)` or `old_acc`, but not in `ustack(p)` or `new_acc`. Conversely, R^+ is the set of universes that become accessible as a result of adding edges from w to the universes that occur in `ustack(p)` or `new_acc`, but not in `old_ustack(p)` or `old_acc`.

This process is implemented by procedure `updateRealm`:

```
proc updateRealm( witness : Universe,
                  old_acc : Set[Universe],
                  new_acc : Set[Universe],
                  p : Process ) is
var garbage, newlinks : Set[Universe];
begin
    garbage := (old_ustack(p) + old_acc) -
        (ustack(p) + new_acc);
```

```
        newlinks := (ustack(p) + new_acc) -
            (old_ustack(p) + old_acc);
        contractRealm(garbage, p);
        completeRealm(witness, newlinks, p);
        old_ustack(p) := ustack(p);
    end
```

The procedure first computes sets `garbage` and `newlinks`, which describe those universes that have become disconnected or have been made immediately accessible, respectively. Procedure `contractRealm` is invoked to compute and remove R^- from the realm and `completeRealm` is invoked to compute and add R^+ to the realm. Finally, the universe stack is saved for the next invocation of `updateRealm`. The remainder of this chapter describes the realm contraction and realm completion procedures in detail.

4.3 Realm contraction

A realm is contracted by *unlinking* edges between universes that, as a result of a realm-affecting operation, are no longer accessible from either the witness or the universe stack. If, after being unlinked, a universe is no longer the target of any accessible edges, then the universe is inaccessible to the process and is thus *released* and made available for migration to other realms. Of course, releasing one universe could unlink others, which may cause more universes to be released. This behavior is reminiscent of the way in which dynamically allocated memory is returned to the heap using reference-counting approaches to garbage collection. Our realm-contraction algorithm borrows from recent results in this area, specif-

ically reference-counting with cycle detection [MWL90, JL96, BAL+01, BAR+03], which we now briefly review.

Simple reference-counting approaches to garbage collection cannot handle cycles of garbage because nodes in a garbage cycle will have non-zero reference counts even when these nodes are not reachable. To handle cycles, a collector must treat each node with a non-zero reference count as a potential root in a garbage cycle. The algorithms of Bacon et al. [BAR+03] detect cycles by performing a depth-first search of each potential cycle root, subtracting the reference counts to see if they go to 0. If the reference count of every visited node goes to zero, then a garbage cycle has been discovered, and every such node can be released. By contrast, the existence of a node with a non-zero reference count refutes the existence of a garbage cycle involving that node; thus any reference counts that were decremented during cycle detection must now be repaired. We adapted these algorithms to the needs of the universe model.

Figure 11 depicts our realm-contraction algorithms. The function `unlink` uses two supporting functions—`dfsDecRef` and `dfsIncRef`—to remove a link to a universe and return the set of universes that need to be released as a result of removing this link. Function `dfsDecRef` performs a depth-first traversal of all universes reachable from a given universe u. When a universe is visited, its reference count is decremented by one, and it is added to a buffer of visited universes, which is returned to the caller. Upon completion, all universes reachable from u will have been visited along every realm-accessible edge. Moreover, the reference count of each visited universe will have been decremented by the number of incoming edges that were traversed.

Function `dfsIncRef` is used to repair the reference counts of universes that are not

members of a garbage cycle. Following `dfsDecRef`, any universe $u \in$ `nodes` with a non-zero reference count is still reachable; thus any universe v reachable from u is not part of a garbage cycle. The set of all such v is discovered by a depth-first traversal, originating at u. A side-effect of this traversal is to increment the reference count of each such v, thus repairing it. Observe that the function will be initially invoked on a universe with a non-zero reference count, and the intention is to increment the reference count of each universe reachable from this source but not increment the count of the universe itself. We achieve this effect using a parameter `inc`, which defaults to 0 when the function is invoked without providing a value for this parameter, but which is always supplied with the value 1 when the function is invoked recursively.

The `unlink` function works in three phases. First, `dfsDecRef` is invoked to decrement the reference count of every universe reachable from the universe referenced by parameter `t` along every reachable edge. These universes are returned in the set `nodes`. Many of these nodes will have reference counts of 0; however, some of these counts may be incorrect, as `dfsDecRef` may have decremented the reference count of a universe whose reference count is unaffected by the unlinking of `t`. The **for** loop discovers and repairs the count of such universes by first identifying those universes `s` $\in$ `nodes` with non-zero reference counts and then incrementing the reference counts of all universes reachable from each such `s` (using `dfsIncRef(s)`). Each invocation of `dfsIncRef` unmarks and returns a set of universes, each of which has a reference count greater than 0, and these universes are removed from `nodes`. Notice that the loop iterates over a copy of the set `nodes` and thus the iteration is unaffected by the removal of unmarked universes from `nodes` in the body of the loop. When the loop terminates, every universe that remains in the set `nodes` will

```
function dfsDecRef( u : Universe )
returns Set[Universe] is
    var visited : Set[Universe] := ∅;
begin
    RC(u) := RC(u) - 1;
    if (¬ Marked(u)) then
        Marked(u) := true;
        insert u into visited;
        for (s ∈ raccessibles(u)) do
            visited := visited
                    ∪ dfsDecRef(s);
        end
    end
    return visited;
end

function unlink( t : Universe )
returns Set[Universe] is
    var nodes : Set[Universe];
    var changed : bool;
begin
    nodes := dfsDecRef(t);
    for (s ∈ nodes) do
        if (RC(s) ≠ 0 and Marked(s)) then
            nodes := nodes - dfsIncRef(s);
        end
    end
    return nodes;
end
```

```
function dfsIncRef( u : Universe,
                    inc : Natural := 0 )
returns Set[Universe] is
    var liveNodes : Set[Universe] := ∅;
begin
    RC(u) := RC(u) + inc;
    insert u into liveNodes;
    if (Marked(u)) then
        Marked(u) := false;
        for (v ∈ raccessibles(u)) do
            liveNodes := liveNodes
                    ∪ dfsIncRef(v, 1);
        end
    end
    return liveNodes;
end

proc releaseAll( nodes : Set[Universe],
          p : Process) is
begin
    for (s ∈ nodes) do
        release(s, p);
    end
end

proc contractRealm(garbage : Set[Universe],
          p : Process) is
begin
    for (g ∈ garbage) do
        releaseAll(unlink(g), p);
    end
end
```

Figure 11: Functions and procedures for realm contraction.

have a zero reference count. This set is returned to the caller.

The function `contractRealm` modifies the realm by first determining which targets of the witness universe should be unlinked and then calling `unlink` on each such universe. The function `releaseAll` merely invokes another function, `release`, on every universe in the set returned by `unlink`. Observe that whereas every function in Figure 11 operates within the realm of a given process, these functions do not require additional synchronization logic. By contrast, `release` modifies resources that are used by realm-completion functions; thus we defer its treatment to the sequel.

Finally, our implementation actually incorporates several optimizations of these algorithms. First, because most realms tend to be trees, `dfsDecRef` is optimized to handle tree structures more efficiently. If the initial universe has a reference count of 1 prior to being marked, then the universe can be released immediately and not inserted into the set returned by `dfsDefRef`. This same optimization can be applied to all children recursively, provided that no child has a reference count greater than 1 prior to marking. In addition, function `contractRealm` optimizes its calls to `unlink` by performing all of the `dfsDecRef` operations over the universes in `garbage` before performing any of the `dfsIncRef` operations. This improves the efficiency of contraction in the case when a cycle spans the set of universes reachable from two or more of the universes in `garbage`.

4.4 Realm completion

A realm is completed by *linking* universes that, as a result of a realm-affecting operation, are now accessible. Linking a universe u may require linking universes that u depends upon

```
proc completeRealm( w : Universe,
                    newlinks : Set[Universe],
                    p : Process ) is
    var u : Universe;
begin
    witness(p) := w;
    pending(p) := 0;
    for (s ∈ newlinks) do
        link(s, p);
    end
    while (¬realmComplete(p)) do
        u := nextAcquired(p);
        bfsLink(u,p);
    end
end

function bfsLink( r : Universe,
                  p : Process ) is
begin
    for (c ∈ accessibles(r)) do
        link(c, p);
    end
end
```

```
proc link( u : Universe,
           p : Process,
           rc : Natural := 1) is
begin
    LOCK(u);
    if (owner(u) = p) then
        RC(u) := RC(u) + rc;
    else
        acquire(u, p, rc);
    end;
    UNLOCK(u);
end
```

Figure 12: Functions and procedures for realm completion.

to satisfy its concurrency constraint. Thus, our algorithms (Figure 12) operate by traversing (breadth first) edges that are accessible from universes in the realm—specifically from a witness universe *w* or from a universe that has appeared on the universe stack since the previous realm update. This initial set of universes to traverse is provided by the parameter `newlinks`, supplied by the call from the procedure `updateRealm`. As with any recursive graph traversal, our algorithm employs a marking strategy, where a universe is considered to be *marked* if it is part of the realm of the process whose realm is being completed.

Function `link` attempts to link a universe `u` using `rc` as the initial reference count for

u.[1] A marked universe is linked by incrementing its reference count; whereas an unmarked universe is linked by *acquiring* it. Note that `link` needs to be atomic to avoid interference by other processes and thus locks the target universe for the duration of its execution.

Because a process may not be able to immediately acquire a universe, acquisition is a two-step protocol of *negotiation*. Figure 13 depicts the different steps of negotiation as states and transitions in an hierarchical labeled-transition system (LTS).[2] Process *p* begins negotiation for a universe *u* by first requesting to acquire *u*, as indicated by the transition acquire. If *u* is being used by another process, then *p*'s negotiation for *u* stalls in the Requesting state, where it cannot proceed until the other process finishes using it. Later, when the process that owns *u* releases it (by calling procedure `release`), that process will check to see if any other processes have requested acquisition for *u* and then explicitly notify one of these requesters. In the sequel, we describe how procedure `acquire` handles this case by registering a request for *u* and then returning, so that *p* may try to acquire other universes while waiting for *u*. In Figure 13, this notification is depicted by the ACQUIRED transition, after which we say that *p owns u*. Note that for simplicity in our algorithms, we follow this two-step negotiation even when a requested universe is not in use by another process. Such acquisitions will also complete with a notification to the requesting process, but in this case the notifying process is the same as the requesting process.

For every call to `acquire`, the caller (i.e., `completeRealm`) is obliged to wait for the notification. This is accomplished by invoking the `nextAcquired` function, which

[1] When invoked from `bfsLink`, the value of `rc` will always be 1. In the sequel, we show that `link` is also used after a process is forced to surrender a universe, which may occur when the universe is the target of multiple links in the realm.

[2] using the UML state-diagram notation [BJR99].

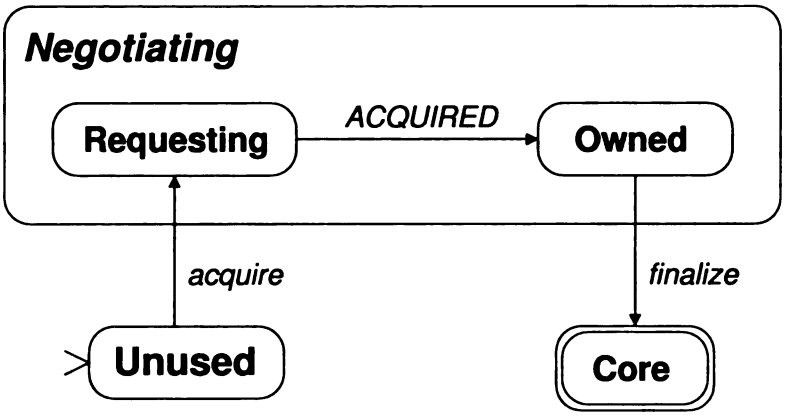

Figure 13: Protocol of negotiation by a process for a universe.

waits for a notification and returns the acquired universe. Once acquired, a universe is considered marked, and the search proceeds by traversing its outgoing accessible edges. This is accomplished by recursively invoking `bfsLink` on the newly acquired universe. After every such edge is traversed, the search terminates, and the realm has been extended by R^+. We coordinate this traversal by recording, for each process, the number of pending acquisition requests (i.e., `pending(p)`), and testing for termination using the function `realmComplete`, which consults this pending attribute.

4.5 Preventing Starvation and Deadlock

As processes negotiate for universes, care must be taken to prevent starvation and deadlock. To this end, we adapt and integrate existing deadlock avoidance strategies and deadlock recovery heuristics to mediate negotiations among processes for universes. Ultimately, all of these algorithms are implemented in terms of the four primitive functions—`acquire`, `nextAcquired`, `realmComplete`, and `release`—used by our realm-update proce-

dure. Because these algorithms collaborate by manipulating data in shared memory, there are many special cases to consider, and thus the algorithms are complex.

A key concept in our understanding and development of these algorithms is the negotiation abstraction, which supports two dimensions of refinement that help to clearly specify and validate our algorithms. The first dimension is a form of *feature refinement* [FK01], whereby a state-based specification is extended with new behavior by adding new states and transitions but never removing any of the existing states or transitions. Using feature refinement, we were able to incrementally extend the basic negotiation protocol in Figure 13 to incorporate:

1. an advanced deadlock-avoidance strategy (Section 5.1);

2. an advanced deadlock-recovery heuristic (Section 6.1); and finally

3. support for negotiating conditional contracts (Chapter 7).

Each of these three refinements involves expanding a state in the less refined specification to include substates and adding a small number of transitions into and among these substates.

In addition to supporting feature refinement, it is also straightforward to apply more traditional *data refinements* that demonstrate how the abstract states and transitions in a negotiation protocol are implemented as concrete data structures and algorithms. In our case, the data structures attribute the `Process` and `Universe` objects to which the negotiation refers, and the algorithms correspond to case logic that fleshes out the four functions we need to implement to complete the suite of realm-update procedures. Observe, in discussions, that we will often refer to an individual negotiation, as if it is an object as opposed to an abstract state of collaboration among multiple objects. We adopt the notation $\langle p, u \rangle$ to

refer to *the* negotiation by process p for universe u. This simplifies presentation by enabling us to talk about a particular negotiation being "in" a given state or witnessing a transition, but bear in mind that negotiation is only an abstraction and that there is no single run-time data structure that localizes this state.

In presenting the refinements, we often need to say that a negotiation is "in" a given state. We denote the current state of a negotiation by adjoining the process and universe identifiers to the name of the name of the abstract state. For example, Owned(p,u) says that negotiation $\langle p, u \rangle$ is state Owned. Most of our data refinements relate predicates over concrete data structures to the abstract states of a negotiation. For example, the notation:

$$\mathsf{Owned}(p, u) \quad \triangleq \quad \texttt{owner}(u) = p \ \wedge \ \neg\texttt{migrating}(u)$$

specifies that negotiation $\langle p, u \rangle$ is in the state Owned if and only if the `owner` attribute of u equals p and the `migrating` attribute of u is false.

We structure the presentation of our algorithms around these two dimensions of refinement. Specifically, each of Chapters 5, 6, and 7 concerns a feature refinement of the negotiation protocol and algorithms of the section that precedes it. Moreover, each section is organized so as to first present the specification of the negotiation protocol at that level of feature refinement followed by a data refinement that shows how to implement that specification. Before continuing, we introduce one final concept that will be used in all of the subsequent refinements.

All deadlock-prevention strategies work by coercing a process to relinquish resources that it has previously acquired. In the universe model, contractual obligations limit the

universes that a process can relinquish. We define the *core* of a process p, written $Core(p)$, to be the universes in $Realm(p)$ that cannot migrate out without violating a contract that is in force prior to (and is unaffected by) the operation that caused the realm to be updated. When a process is executing user code, $Core(p) = R_p$; whereas when a process is updating its realm, $Core(p) = R_p \setminus R_p^-$, i.e. the set of universes that remain in the realm of the process after realm contraction. Also, note that our negotiation protocol represents the membership of a universe in the core of a process. Specifically, when a process p completes its realm (i.e., when the function `realmComplete` returns true), all negotiations for universes in R_p^+ will *finalize*, which means they all simultaneously transition into the final state **Core**.

Chapter 5

Feature: Deadlock Avoidance

Our first refinement extends the negotiation protocol to allow multiple concurrent processes to acquire multiple (non-core) universes while avoiding starvation and deadlock. The procedure borrows ideas from a resource-allocation strategy called *wound–wait*, under which processes can preemptively steal resources allocated to other processes [RSL78]. The idea is that when two or more processes attempt to acquire the same universe, priority is given to the oldest process. We say that a process p is older than another process q if p began its most recent realm-update operation earlier than q. Suppose, without loss of generality, that p is older than q. If q attempts to acquire a universe currently held by p, then q *waits* for p. Conversely, if p attempts to acquire a universe currently held by q, then p *wounds* q by signaling q to surrender the universe. Following this scheme, and provided that processes compete only for non-core universes, the oldest process in the system will eventually be able to acquire all the universes it needs and thus complete its realm. Successively, the same then applies to the second-oldest process, the third-oldest process, and so on.

5.1 Specification

Based on this strategy, we refined the negotiation protocol to allow processes to communicate by exchanging messages with one another. Under this protocol, inter-processes communication is asynchronous, and there are two kinds of messages—ACQUIRED and SURRENDER, both of which are parameterized by a reference to the universe under contention. Intuitively, a process p sends a message of the form ACQUIRED (u) to a process q to signal that q is the new owner of u. Observe that p and q may be the same process, which might happen if, when attempting to acquire a universe, the process discovers that there are no contending processes. In contrast to signaling acquisition, a process p sends a message of the form SURRENDER (u) to a process $q \neq p$ to signal q to relinquish ownership of u.

Figure 14 refines the negotiation protocol to incorporate the wound–wait strategy. Requesting is now an hierarchical state with two sub-states. Upon entering Requesting(p,u), if no other process can legally claim ownership of u, we say u is *migrating* to p; otherwise we say that p is *waiting* for u. Entry into Migrating(p,u) coincides with a process sending an ACQUIRED (u) message to p. Thus, migrating implies that there are no obstacles to p acquiring u other than for p to actively receive and handle the message, and negotiation transitions to Owned once the message is handled. By contrast, Waiting(p,u) implies that there are obstacles to p acquiring u, e.g., if another process currently owns u.[1] Observe that when Migrating(p, u) is entered from Unused or Owned, p is both the sender and the receiver of the ACQUIRED (u) message. However, if the state is entered via the enable

[1] There is another kind of obstacle, namely that p is negotiating a conditional contract for u and the condition cannot be verified. At this level of feature refinement, we are only concerned with the negotiation of exclusion contracts. We deal with the unverified condition obstacle in Chapter 7.

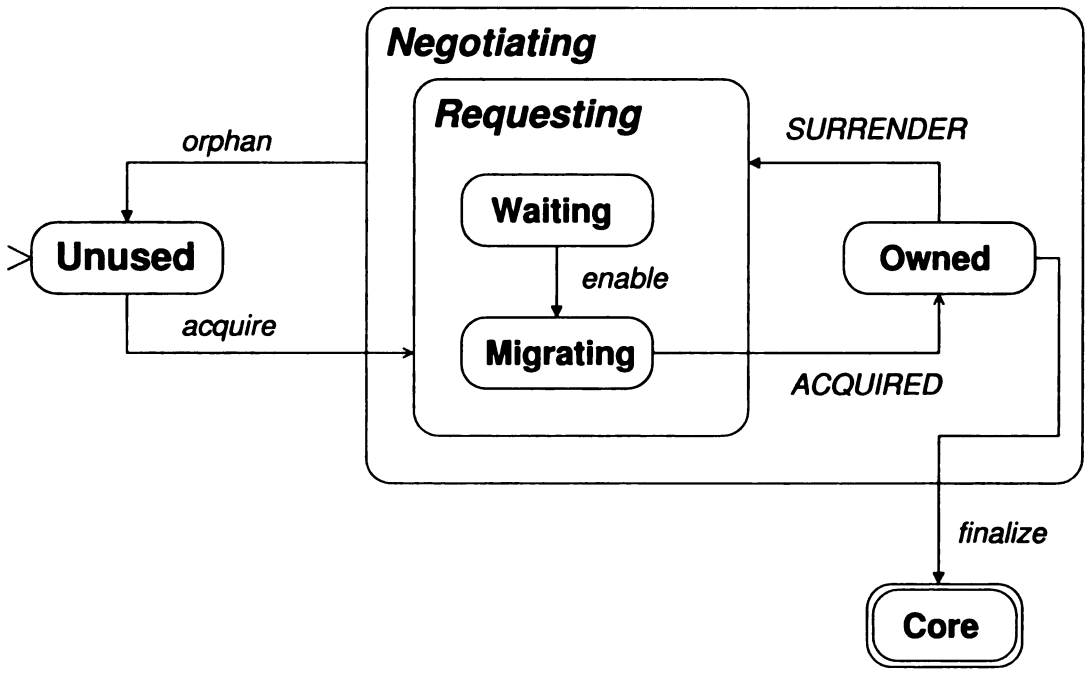

Figure 14: Refinement of protocol in Figure 13 to avoid deadlocks using wound–wait strategy.

transition, then the message will come from a process other than p.

The refinement includes two transitions to implement the wounding of one process by a higher-priority process. Suppose a process p attempts to acquire a universe u that is owned by another process q, such that $u \notin Core(q)$. If p is older than q, then p will send a SURRENDER (u) message to q coincident with entry into Waiting(p,u).[2] The SURRENDER(q,u) transition is taken when q receives and acts on this message. In addition to causing q to release ownership of u, this transition can engender orphan(q, v) transitions in other negotiations $\langle q, v \rangle$ where $u \neq v$. This can happen, for example, if v is accessible from u and the release of u causes v to be released. We refer to this process as *orphaning* a universe and note that a universe could be orphaned at any time during

[2]This phenomena is not depicted in the diagram.

negotiation.

Finally, the negotiation protocol assumes some inter-negotiation synchronizations that are not depicted in the diagram. For example, the transition SURRENDER(q, u) is coincident with the transition enable(p, u) when q releases universe u and p, which was waiting for u is sent the `ACQUIRED` (u) message. Note that sending the `ACQUIRED` message is not depicted in the diagram. A similar synchronization occurs when p is waiting for $u \in Core(q)$, and q executes a realm-affecting operation that causes it to release u (during realm contraction). In the sequel, we will refine the responsibility for such inter-process synchronizations into operations over universes (which are passive objects) rather than processes, which only communicate asynchronously. Finally, all of the negotiations for a given process must synchronize on the finalize transition. In the sequel, we refine the responsibility for this intra-process synchronization into an atomic operation on the the process object, without reference to any of the universe objects.

5.2 Data Refinement

We now describe the primitive data structures and functions used to implement the protocol presented in Section 5.1. One category of primitives refines the abstract states of negotiation into concrete data structures that can be queried and manipulated by our algorithms (Section 5.2.1). Another category of primitives implements the message-passing infra-structure (Section 5.2.2); while a third describes some additional book-keeping attributes that are necessary for advanced deadlock avoidance. For brevity, all of the functions introduced in this section appear in the appendix.

5.2.1 State refinements

Our specification (Section 5.1) refers to several concepts that are used to define observable states in the protocol. These include process priority, the core of a realm, and the various negotiation sub-states in Figure 14. We refined these concepts into concrete data structures that attribute processes and universes, thus allowing efficient representation and observation of these abstract states.

Both process priority and membership of a universe in the core of its realm are represented using time stamps. Each process contains a *start time*, which is the time when the process began its most recent realm-update operation. The relative age of two processes can be established by comparing these times, e.g., if `start_time(`p`)` is less than `start_time(`q`)`, then p is older than q. Likewise, each universe contains an *acquisition time*, which records the start time of its owner at the time the owner began to migrate the universe into its realm. To test if a universe is in the core of its owner's realm, one can simply check if the acquisition time of the universe is less than the start time of its owner. When this check returns true, we can infer that the universe was acquired by a previous realm-update operation, which means it must be in its owner's core.

The start time of a process is also used to order processes that are competing to acquire the same universe. We associate with each universe a priority queue with which to buffer requests for acquisition. Each queue element is an ordered pair that comprises a process identifier and a reference count. Elements are ordered by the start time of the process coordinate, such that the head of the queue is the oldest process (i.e. the process p with the lowest value of `start_time(`p`)`). We use the attribute `reqQ(`u`)` to access the priority

queue associated with universe u. In addition to the standard `enqueue` and `dequeue` operations, these priority queues support an operation `contains`, which determines whether a given process is waiting inside the queue, and an operation `updateQueueRC`, which adjusts the reference count associated with a process that is waiting in the queue.

Finally, our algorithms must distinguish among the various states of negotiation, as depicted in Figure 14. We distinguish between states **Migrating** and **Owned** by associating a boolean attribute `migrating`, with each universe u. This attribute is set when a universe begins to migrate into the realm of a process and reset once the process receives the `ACQUIRED` message and completes the migration. Negotiation is in the **Waiting** state if a process is in the request queue of the universe. More formally:[3]

$$\textbf{Migrating}(p,u) \triangleq \texttt{owner}(u) = p \ \wedge \ \texttt{migrating}(u)$$

$$\textbf{Owned}(p,u) \triangleq \texttt{owner}(u) = p \ \wedge \ \neg\texttt{migrating}(u)$$

$$\textbf{Waiting}(p,u) \triangleq \texttt{contains}(\texttt{reqQ}(u), p)$$

5.2.2 Inter-process messaging

We implement inter-process messaging using FIFO queues and two procedures—`signal` and `getMessage`. A process sends a message to a receiver process by invoking procedure `signal`, which is parameterized by the process, the message and the receiver. Messages are stored in a message queue associated with the receiver. The message queue of a process p is accessed using the attribute `msgQ` (p). Function `getMessage` retrieves the next

[3]The pair (`owner` (u), `migrating` (u)) must be tested and set atomically in order to avoid race conditions. This can be accomplished by encoding the process identifier of the owner and the migrating flag in a single machine word.

message in the associated process' queue, blocking if the queue is empty.[4]

To monitor the progress of realm completion, we attribute each process with the number of *pending requests* for universes that the process has requested but does not yet own. The value of `pending(`p`)` is incremented each time p attempts to acquire an unused universe and decremented each time p successfully negotiates ownership of a new universe. Once `pending(`p`)` reaches zero and all messages have been processed (i.e., `msgQ(`p`)` is empty), then the realm is complete.

Before concluding, note that because our protocol is asynchronous, messages might become stale while they are waiting to be processed. A message `ACQUIRED(`u`)` (or `SURRENDER(`u`)`) grows stale when, in the interval between when the message is sent and when it gets processed, the receiver is forced to surrender u from its realm. Observe that a process might surrender a universe indirectly, as a side-effect of being forced to surrender another universe. Suppose, for example, that a process receives a `SURRENDER(`v`)` message and that there is a realm-accessible edge $v \rightarrow u$ to another universe u. Then releasing v will decrement the reference count of u, and if this count goes to zero, u must be released. If u is released, then any pending messages of the form `ACQUIRED(`u`)` or `SURRENDER(`u`)` are stale. We handle stale messages by explicitly checking for staleness whenever we retrieve a message from the queue.

[4]Pseudocode for both of these procedures appears in the appendix. In Section 6.3, we refine the `getMessage` function to incorporate deadlock recovery.

5.2.3 Additional primitives

Finally, the attribute `blocked(p)` contains the set of all universes on which `p` is blocking indefinitely. A process is blocking indefinitely on a universe if that universe is in the core of another process or has been requested by an older process. The process is blocking indefinitely in this case because it requires an action by the other process to release the universe. A universe is added to this set by calling `recordBlocked(p)` and removed from it by calling `recordUnblocked(p)`. Note that while the value of `blocked(p)` is not necessary for the basic algorithm, it will be used in the advanced deadlock avoidance scheme to determine potential deadlock.

5.3 Algorithms

We defined the universe-acquisition protocol from the point of view of a given process to be the orthogonal composition of independent negotiations associated with that process. The `completeRealm` algorithm (Figure 12) simulates the composition of these independent negotiations by successively issuing requests to acquire universes and then looping over the acknowledgements of these acquisitions (possibly issuing new requests) until, by some measure, the realm is determined to be complete. We use this (event-driven) sequential control structure to simulate the concurrent execution of negotiations within a given process. Moreover, we assign responsibility for implementing the various negotiation transitions (Figure 14) to the procedures `acquire`, `nextAcquired`, and `realmComplete`, which are used as primitives in the `completeRealm` algorithm (Section 4.4).

We present these algorithms by first defining `acquire` and `nextAcquired` (Sec-

tion 5.3.1). In terms of the negotiation protocol, `acquire` implements the **acquire** transition, and `nextAcquired` implements **ACQUIRED** and **SURRENDER** for negotiation $\langle p, u \rangle$. By contrast, function `completeRealm` implements the simultaneous transition **finalize(*p*,*u*)** for all u for which there exists a negotiation $\langle p, u \rangle$, using function `realmComplete` to decide when it is appropriate to take the transition. The implementation of `realmComplete` is trivial and appears in the appendix.

To implement **SURRENDER** in `nextAcquired`, we use an auxiliary function called `surrender` (Section 5.3.2). This procedure implements the **orphan** transition. Procedure `surrender` also uses the `release` procedure, which we first referenced in `contractRealm`. The algorithm for release (Section 5.3.3) implements the **enable** transition, thus completing the algorithmic refinement of the negotiation protocol.

5.3.1 Universe Acquisition

Universe acquisition is a multi-step process, initiated by invoking the procedure `acquire` and completed by invoking the function `nextAcquired`, both of which are depicted in Figure 15. Procedure `acquire` is invoked with three parameters—a reference `u` to the universe to be acquired, a reference `p` to the process wishing to acquire `u`, and a reference count `rc` with which to set `u` once it becomes part of the realm of `p`. Negotiation then begins, entering the **Requesting** state, in which there are two cases to consider. If `u` has no owner, then it is available for migration, in which case migration is initiated by calling `enterMigrating`, which sets the `owner`, `RC`, `acquisition_time`, and `migrating` attributes, and then sends an `ACQUIRED(u)` message to `p`. Alternatively,

```
proc acquire( u : Universe,
              p : Process,
              rc : Natural ) is
begin
    if (owner(u) = ⊥) then
        pending(p) := pending(p) + 1;
        enterMigrating(u,p,rc);
    else
        enterWaiting(u,p,rc);
    end;
end

proc enterMigrating( u : Universe,
                     p : Process,
                     rc : Natural ) is
begin
    Marked(u) := false;
    migrating(u), owner(u) := true, p;
    RC(u) := rc;
    acquisition_time(u) := start_time(p);
    LOCK(p);
    signal(p, ACQUIRED, u);
    UNLOCK(p);
end

proc enterWaiting( u : Universe,
                   p : Process,
                   rc : Natural ) is
begin
    request(u,p,rc);
    if (¬ wound(u,p)) then
        recordBlocked(u,p);
    end
end

proc enterOwned( u : Universe,
                 p : Process ) is
begin
    migrating(u) := false;
    pending(p) := pending(p) - 1;
    recordUnblocked(u, p);
end
```

```
function nextAcquired( p : Process )
returns Universe is
    var msg : MessageType;
    var u : Universe;
begin
    loop
        ⟨msg, u⟩ := getMessage(p);
        if (msg = ACQUIRED) then
            if (inMigrating(u,p)) then
                enterOwned(u, p);
                return u;
            end
        else /* msg = SURRENDER */
            if (inOwned(u,p)) then
                surrender(u, p);
            end
        end
    end
end

function wound( u : Universe,
                p : Process )
returns boolean is
    var wounded : boolean := false;
begin
    if ( start_time(p) < acquisition_time(u)
        ∧ p = head(reqQ(u)) )
    then
        LOCK(owner(u));
        if (u ∉ Core(owner(u)))
        then
            signal(owner(u), SURRENDER, u);
            wounded := true;
        end
        UNLOCK(owner(u));
    end
    return wounded;
end
```

Figure 15: Functions and procedures for acquiring a universe while avoiding deadlock.

if u has an owner, then negotiation enters the Waiting state, which is accomplished by invoking the procedure `enterWaiting`.

Within `enterWaiting` there are two cases to consider: Either p has priority over the current owner of u, in which case p can preemptively steal u from its owner, or p must block, waiting for the current owner (and possibly other competing processes of higher priority) to release u. In either case, p must register in the priority queue of u, which is accomplished by invoking the procedure `request`, whose implementation appears in the appendix. After registering p with u, p then attempts to wound the owner of u. If p lacks the priority to wound the owner of u, then p must wait for the owner to release u. In this case, we say that p is *blocked* on u, and we record this fact by invoking the function `recordBlocked`. This information will be used in the version of these algorithms that incorporates deadlock recovery (Chapter 6).

Function `wound` first checks to see if the process p can legally preempt u from its owner. For the preemption to be legal, (1) p must have higher priority than the owner of u; (2) p must be at the head of the request queue of u; and (3) u must not be in the core of its owner's realm. The pseudo-code incorporates some optimizations. If p has higher priority than the owner of u, then the start time of p must be less than the start time of the owner of u. This condition can be efficiently checked by comparing the start time of p against the acquisition time of u, because the latter value equals the start time of `owner(u)`, unless u is in the core of its owner's realm. Observe that `owner(u)` must be locked so that the test for u being in the core and the subsequent signal to surrender are atomic. If conditions (1)–(3) are satisfied, then p signals the owner to surrender u. If any of these checks fails, function `wound` returns false.

Function `nextAcquired` is responsible for handling messages sent to a process. Each call to this function retrieves and returns the next `ACQUIRED` message in the calling process' message queue. To find an `ACQUIRED` message, `nextAcquired` might encounter zero or more `SURRENDER` messages, each of which are handled without returning from this function. If the queue becomes empty prior to finding an `ACQUIRED` message, then `nextAcquired` will block.

`ACQUIRED` messages are meant to trigger a state transition from **Migrating** to **Owned**. To handle such messages, `nextAcquired` first checks to see if the current state of negotiation is **Migrating** (by invoking function `inMigrating`). If so, negotiation enters the **Owned** state (by invoking `enterOwned`), and u is returned to the caller of `nextAcquired`. Observe that the check is necessary because, as described in Section 5.2.2, the message could have become stale while it was waiting to be serviced. By the same reasoning, `nextAcquired` must also check that `SURRENDER(`u`)` messages are not stale. Commensurate with the protocol, such a message is only meaningful when negotiation is in the **Owned** state, which we check using the function `inOwned` (defined in the appendix).

Finally, procedure `enterOwned` completes the migration process by setting `migrating(u)` to false and recording that `p` no longer blocks on `u`, if that was the case before. In addition, the count of pending acquisition requests is decremented.

```
proc surrender( u : Universe,
                p : Process ) is
var rc : Natural;
    unlinked : Set[Universe];
begin
    if (¬ migrating(u)) then
        for (s ∈ raccessibles(u)) do
            unlinked := unlink(s);
            releaseFringe(unlinked, p);
            releaseAll(unlinked, p);
        end
        releaseAll(accessibles(u) -
                raccessibles(u), p);
    end
    rc := RC(u);
    release(u, p, rc);
    link(u, p, rc);
end
```

```
proc releaseFringe( nodes : Set[Universe],
            p : Process) is
begin
    for (s ∈ nodes) do
        releaseAll(accessibles(s) -
            raccessibles(s), p);
    end;
end
```

Figure 16: Procedures for surrendering a universe to another process.

5.3.2 Universe Surrender

Function `nextAcquired` delegates responsibility for handling the transition from **Owned** to **Requesting** to a procedure called `surrender`, which causes a process p to release and then immediately re-request a universe u. Once u is released, a higher-priority process will be able to acquire it, and p will then wait for u to become available again. Of course, if p owns or is currently requesting other universes that are accessible only from u, then these other universes must also be released. Thus, surrendering a universe is similar to removing all incoming accessible links to the universe; however, there are two important differences. First, the reference count of u prior to the surrender must be stored somewhere so that `RC(`u`)` can be properly reestablished once p re-acquires u. Second, and most important, the algorithms developed in Chapter 4 for recursively releasing inaccessible universes

only traverse links between universes that are realm accessible (i.e., both the source and the target of the link is in the realm). However, when p is instructed to surrender u, p might be negotiating for universes that are accessible but not yet realm-accessible from u. Thus, the surrender algorithm must handle this case separately.

Figure 16 depicts the algorithm for procedure `surrender`, which releases and then immediately relinks a universe `u`. Notice that `surrender` may be invoked when $\langle p, u \rangle$ is in state **Migrating** or state **Owned**. While no action other than releasing and relinking `u` is necessary in state **Migrating**, in state **Owned**, `p` will have already requested universes accessible from `u`. Moreover, the realm of `p` might comprise accessible edges that are not yet realm accessible, because the targets of these edges have not yet been acquired. We refer to the targets of such edges collectively as the *fringe* of the realm. Because `unlink` only traverses realm-accessible edges, the set of universes that it returns will not include fringe universes. Thus we must compute (and then release) these fringe universes separately. The procedure `releaseFringe` computes and releases fringe universes that are accessible from a set of universes that are known to be in the realm, after which all accessible edges originating at `u` will have been unlinked. Finally, `u` is released and then immediately relinked.

Figure 17 contains a step-by-step example that illustrates the process of surrendering a universe. In this figure, black circles denote *owned* universes, gray circles denote *requested* universes (the fringe), and white circles denote *unused* universes. Each universe has an associated reference count listed next to it. Arrows denote accessible edges. Arrows become dashed when they are unlinked (in function `dfsDecRef` or `release`) and the reference count of the target universe is decremented. Dashed arrows become solid again when the

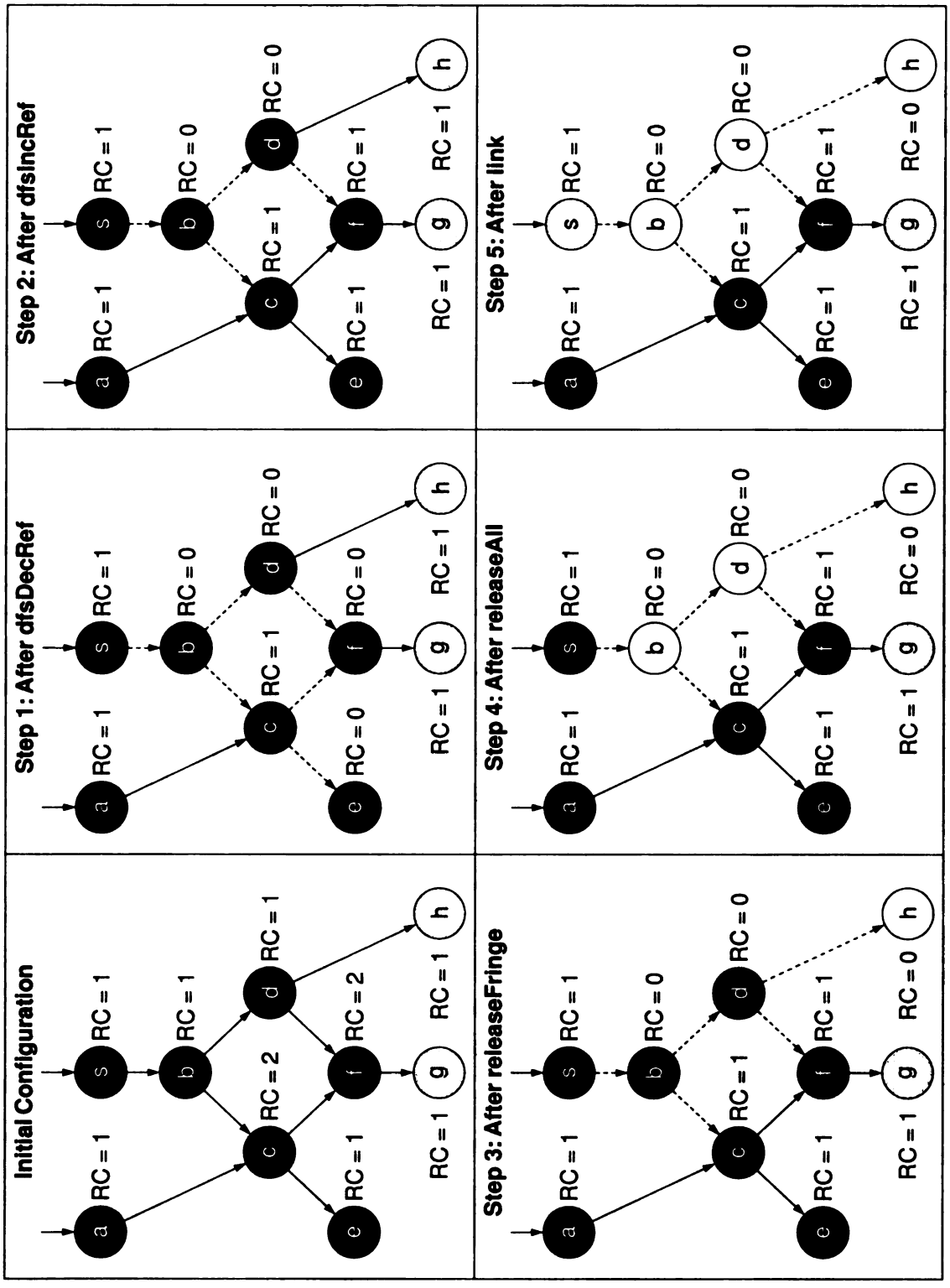

Figure 17: Example: Surrendering a universe

link is reestablished (in function `dfsIncRef`) and the reference count of the target universe is incremented.

The example begins with the process receiving a `SURRENDER` message for universe s. Because s has a realm-accessible edge $s \rightarrow b$, function `unlink` is invoked on b, which then results in `dfsDecRef` being called on b, as depicted in Step 1. Observe that `dfsDecRef` does not traverse the edges $d \rightarrow h$ or $f \rightarrow g$, since the target universes are not in `raccessibles(d)` or `raccessibles(f)`, respectively. Thus the set returned by `dfsDecRef(`b`)` is $\{b, c, d, e, f\}$. Out of these universes, only c has a reference count > 0. Thus, in step 2, `dfsIncRef` is called on c, which reestablishes the edges $c \rightarrow e$ and $c \rightarrow f$. At this point, b and d are the only universes that are set to be released. In Step 3, however, the call to `releaseFringe` on the set $\{b, d\}$ releases universe h. Step 4 then releases the universes b and d. Finally, in step 5, s is released, followed by an immediate re-acquisition attempt via `link`, making it a requested universe.

5.3.3 Universe Release

Both `contractRealm` and `surrender` use the utility procedure `release` (Figure 18) which, when invoked on a universe `u`, decrements the reference count of `u` and migrates `u` out of the realm if appropriate. There are three cases to consider: Either $\langle p, u \rangle$ is in state Waiting, Migrating, or Owned. If waiting, then `p` is not the owner of `u`, and the only action to take is to update the entry in the request queue of `u`. This update is accomplished by procedure `releaseFromQueue`, which appears in the appendix. Alternately, if waiting or owned, then `p` is the owner of `u`. In either case, we decrement the reference count of `u`,

```
proc release( u : Universe,
              p : Process,
              rc : Natural := 1 ) is
begin
   LOCK(u);
   if (owner(u) ≠ p) then
      releaseFromQueue(u, p, rc);
   else
      RC(u) := RC(u) - rc;
      if (RC(u) ≤ 0) then
         if (migrating(u)) then
            pending(p) := pending(p) - 1;
         end
         migrateNext(u);
      end
   end;
   UNLOCK(u);
end
```

```
proc migrateNext( u : Universe ) is
var newOwner : Process;
   newRC : Natural;
begin
   if (¬ empty(reqQ(u))) then
      ⟨ newOwner, newRC ⟩ :=
         dequeue(reqQ(u));
      enterMigrating(u, newOwner, newRC);
   else
      owner(u) := ⊥;
   end
end
```

Figure 18: Procedures for releasing a universe from a realm.

and if it goes to zero, then u must be migrated out of the realm. This is accomplished by invoking procedure `migrateNext`. If $\langle p, u \rangle$ was in state **Migrating** on entry to `release`, then the number of pending requests in p must be decremented to reflect the release of u.

Procedure `migrateNext` is responsible for actually migrating a universe out of the realm of its current owner and into the realm of a new owner if another process is waiting to acquire it. If there are waiting processes, the function retrieves a new owner and reference count from the request queue and initiates migration to this new owner by invoking `enterMigrating`. If the queue is empty, then the owner is set to $\perp$ to denote that the universe is unused.

Chapter 6

Feature: Deadlock Recovery

We now refine the deadlock-avoiding protocol with capability that prevents a larger class of deadlock situations. Suppose that processes p and q are contending for a universe v, that q is also contending for a universe $u \in Core(p)$, and that q is older than p. Then p can neither release u nor preempt q to acquire v, and the resulting deadlock cannot be avoided using the wound–wait strategy. Our next refinement incorporates a heuristic that can automatically recover from deadlocks in which, for any processes p and q such that q is older than p:

$$R_p^+ \cap R_q^+ \neq \varnothing \quad \wedge \quad R_q^+ \cap Core(p) \neq \varnothing \quad \wedge \quad R_p^+ \cap Core(q) = \varnothing$$

That is, when there is at least one universe that both p and q need to acquire, the universes that q seeks to acquire may overlap with p's core, but not vice versa. The heuristic borrows ideas from an approach that was first informally introduced in [BBD82] and later refined under the name *cautious waiting* [HZ92]. Under cautious waiting, a process that requests a resource will wait for the owner of the resource only if this owner is not also waiting.

Should the owner be waiting, the requestor will *restart* its operation by first releasing and then attempting to re-acquire its resources. Once released, these resources can be acquired by other processes, thereby breaking the deadlock. We apply this heuristic to recover from deadlocks caused when a high-priority process (the requestor) attempts to acquire universes in the core of a lower-priority process (the owner).

6.1 Specification

To add cautious waiting to our protocol required precisely formulating a *restart behavior* and a *restart condition* that triggers this behavior. Because restarting a process is expensive, we designed the condition to forestall triggering the behavior as long as possible, relying on avoidance capability to prevent the majority of deadlocks. This condition optimistically assumes a process is not waiting for another waiting process, barring substantial evidence to the contrary. We formalized the condition into a *direct waiting predicate*, that can be refuted by any negotiation in which a process is engaged, and we formalized the behavior as the simultaneous transition of all such negotiations. Section 6.1.1 refines the individual negotiation protocol with new states and transitions that simplify the specification of the predicate and the effect of the behavior. The direct-waiting predicate is an approximation of a more intuitive condition (Section 6.1.2), which is difficult to compute efficiently. We show how the direct-waiting predicate approximates this more intuitive condition to yield an effective and efficient test (Section 6.1.3)

6.1.1 Negotiation Refinement

Figure 19 refines the negotiation protocol to support checking the restart condition and effecting the restart behavior. Waiting is now an hierarchical state with two sub-states. Upon entering Waiting(p, u), if no process with higher priority is negotiating for u and $u \notin Core(owner(u))$, we say that p is *preempting* the owner of u; otherwise, we say p is *blocking* on u. A preempting negotiation can reasonably expect to transition into Migrating in the near term; whereas a blocking negotiation could block for some indeterminate amount of time. In the sequel, the existence a preempting negotiation is sufficient (but not necessary) to refute the restart condition.

Restarting a process p consists of p: (1) releasing any universe u such that Negotiating(p, u), (2) artificially lowering its priority by updating its start time, and then (3) attempting to reacquire all of its released universes. This operation can break a deadlock by making all of p's universes available to competing processes while preventing p from immediately re-acquiring them by preemption. We refer to p's release of and low-priority re-request for a universe u as *reordering* the negotiation $\langle p, u \rangle$.

We specify the reordering of a negotiation using a new transition called reorder, which transitions into Negotiating. While it might seem that reorder(p, u) should transition into Waiting, there are cases in which, after reordering, $\langle p, u \rangle$ could be either migrating or preempting. If u is not under contention by other processes then p will reacquire it despite its lowered priority. Alternately, because restart is not an atomic operation, other processes could complete their realms and then release a universe that was under contention prior to the completion of the restart operation.

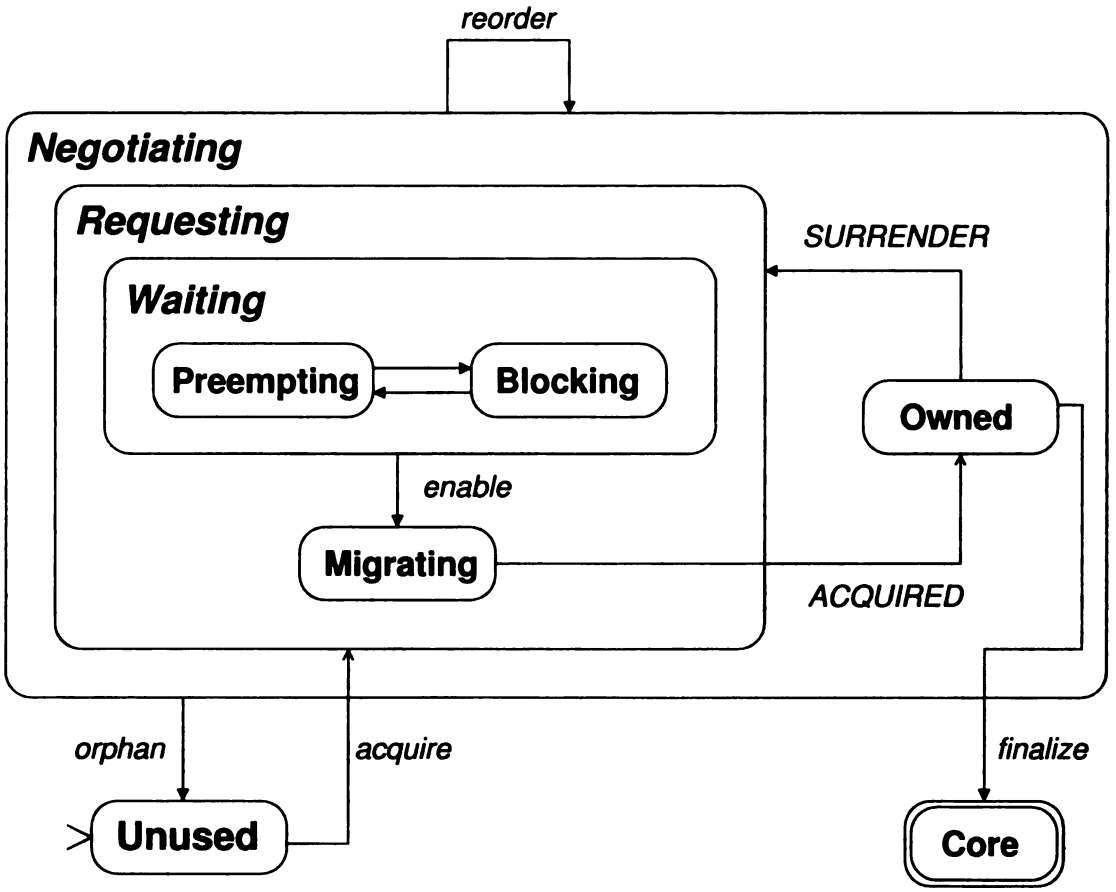

Figure 19: Refinement of protocol in Figure 14 with cautious waiting for deadlock recovery

6.1.2 Restart Condition: Conceptual Definition

We define the restart condition in terms of a process's *belief* in its ability to continue making progress in negotiation. To make the condition precise requires some new terminology. Let p and q be distinct processes, and let W be a set of universes. We say *p directly waits for W relative to q*, if:

1. $W \neq \varnothing$

2. p has requested all of the universes in W but has no reason to believe it can continue negotiating for them until they are released by q; and

3. there is no smaller set $W' \subset W$ that satisfies conditions 1 and 2.

If such a W exists, we refer to it as p's *wait-for set* relative to q. For example, suppose p requests a set U of universes. Then p's wait-for set relative to q, will contain $U \cap Core(q)$ and any universes in U for which q is a contender with higher priority. As this example illustrates, the wait-for sets of a given process need not be disjoint.

Likewise, we say *p indirectly waits for* for W relative to q if:

1. p directly waits for W relative to q; and

2. there exists a set X and a process r such that q directly waits for X relative to r.

Observe that p and r might refer to the same process, which demonstrates indirect waiting is necessary (but not sufficient) for even the simplest form of (unavoidable) deadlock. Our reformulation of the cautious waiting heuristic can therefore be stated as follows:

Indirect waiting heuristic. Restart any process p whose set of outstanding requests is the union of wait-for sets relative to processes for which p is indirectly waiting.

In the sequel, we refer to the condition that triggers the restart as the *indirect waiting condition*.

Checking the indirect-waiting condition is complicated by two factors. First, it could require the run-time system to check if the owner of each requested universe is directly-waiting on another process, a potentially expensive operation if the number of outstanding universes is large. Second, even if the inspection of each universe is inexpensive, the inspection of the whole set is not an atomic operation. To make the check atomic would add to the complexity and expense of the overall algorithm. To avoid such potentially complex and expensive operations, we instead implement a safe approximation of the indirect-waiting condition in terms of sustained direct waiting over a time interval.

6.1.3 Restart Condition: Effective Definition

We approximate the check that a process p indirectly waits for some indeterminate process if there exists a process q, such that p directly waits for q and has continued to do so for some (heuristically determined) time $T \in \mathbb{R}^+$. This approximation works because direct waiting that is not caused by a deadlock will be resolved in finite time; whereas direct waiting due to deadlock will persist until the deadlock is broken. Thus, our approximation of the indirect-waiting heuristic can be stated as follows:

Direct waiting heuristic. Restart any process p whose set of outstanding requests is the union of wait-for sets relative to processes for which p has been directly waiting for some time interval $T \in \mathbb{R}^+$.

In the sequel, we refer to the condition that must persist for time T as the *direct-waiting condition* and denote it mathematically via the function:

$$\mathit{directWaiting} : \text{Process} \rightarrow \text{Bool}$$

The direct-waiting condition is a safe approximation of the indirect-waiting condition in the sense that the former will detect any indirect waiting that persists for at least T time units. Indirect waiting that does not persist for at least T time units could not have contributed to an unavoidable deadlock. On the other hand, the approximation can yield false positives, which incur unnecessary restart operations. While not desirable, false positives are safe because a restart cannot introduce any new direct or indirect waiting conditions, and thus no new deadlocks.

To precisely formulate the direct-waiting condition, we analyze the state space of negotiation to determine the different ways in which a given negotiation can refute the condition. Then, using the pigeonhole principle, we define $\mathit{directWaiting}(p)$ to be the absence of any of these refuting cases. Let U be the set of universes with which a requesting process p is negotiating, and let $P \subseteq U$ be the set of pending requests, i.e., the set of universes that p has requested but does not yet own. Certainly, if P is empty, then p cannot be directly waiting, i.e.:

$$P = \varnothing \Rightarrow \neg \mathit{directWaiting}(p) \tag{6.1}$$

Suppose further that P is not empty, and consider a universe $u \in P$. Then the negotiation $\langle p, u \rangle$ can be in only one of three states—**Preempting**, **Blocking**, or **Migrating**. If preempting or migrating, p has reason to believe it can continue negotiating for u. More formally:

$$(\exists u : P \bullet \textsf{Migrating}(p, u)) \Rightarrow \neg \mathit{directWaiting}(p) \tag{6.2}$$

$$(\exists u : P \bullet \textsf{Preempting}(p, u)) \Rightarrow \neg \mathit{directWaiting}(p) \tag{6.3}$$

Thus, for $\mathit{directWaiting}(p)$ to be true, it is necessary (but not sufficient) for the negotiation corresponding to every outstanding request be in state **Blocking**.

A negotiation $\langle p, u \rangle$ in **Blocking** can only transition out of blocking via an `orphan` or an `enable` transition. Thus, for p to believe it can continue negotiating for u, p must believe that one of these transitions is imminent. There is really no way to predict when a blocking negotiation will be enabled, as this timing is a function of the computation being performed by the owner of u and also the number of contending processes with priority over p. By contrast, the `orphaned` transition may occasionally be predictable. Recall that $\langle p, u \rangle$ can be orphaned if p is negotiating for another universe v, from which u is reachable, and $\langle p, v \rangle$ undergoes a **SURRENDER** transition. Always the optimist, we treat the occurrence of a **SURRENDER** transition as a potential generator of **orphan** transitions in other negotiations. Thus, if there were some way to detect that a **SURRENDER** transition was imminent, such would refute the direct-waiting condition. More formally:

$$(\exists u : U \setminus P \bullet \mathit{imminentSurrender}(p, u)) \Rightarrow \neg \mathit{directWaiting}(p) \tag{6.4}$$

Because the direct waiting condition is defined in terms of universes that p has requested, but does not yet own, equations 6.1. 6.2, 6.3, and 6.4 cover all of the possible refutations of $directWaiting(p)$. That is:

$$\begin{aligned} \neg directWaiting(p) \quad &\Leftrightarrow \quad P = \varnothing && (6.5) \\ &\vee \quad \exists u : U \setminus P \bullet imminentSurrender(p, u) && (6.6) \\ &\vee \quad \exists v : P \bullet \mathsf{Migrating}(p, v) && (6.7) \\ &\vee \quad \exists w : P \bullet \mathsf{Preempting}(p, w) && (6.8) \end{aligned}$$

We call the negation of this formula the direct-waiting predicate, but in order to simplify the data refinement below, we delay this negation.

6.2 Data Refinement

We now apply several identities and logical manipulation to refine the direct-waiting predicate into an efficient predicate over the attributes of a process object. Observe first that a SURRENDER(p, u) transition is imminent if there is a `SURRENDER(`u`)` message waiting on the message queue of p. More formally:

$$imminentSurrender(p, u) \Leftrightarrow \texttt{SURRENDER}(u) \in \texttt{msgQ}(p) \qquad (6.9)$$

Moreover, a negotiation $\langle p, u \rangle$ transitions to Migrating coincident with the insertion of an `ACQUIRED(`u`)` message onto the message queue of p. By combining this knowledge with

lines 6.6 and 6.7, and by duplicating the disjunct in line 6.7 and rewriting it alongside line 6.8, we get:

$$\begin{aligned} \neg \mathit{directWaiting}(p) \;\Leftrightarrow\; & P = \varnothing \\ \vee\; & \texttt{size(msgQ(}p\texttt{))} \geq 1 \\ \vee\; & \exists v : P \bullet \mathsf{Migrating}(p, v) \vee \mathsf{Preempting}(p, v)\} \end{aligned}$$

Then by negating both sides, we get:

$$\begin{aligned} \mathit{directWaiting}(p) \;\Leftrightarrow\; & \texttt{empty(msgQ(}p\texttt{))} \\ \wedge\; & P \neq \varnothing \\ \wedge\; & \forall v : P \bullet \mathsf{Blocking}(p, v) \end{aligned}$$

Finally, observe that algorithms developed in the previous section support the blocking–preempting distinction, as we associate with each process p a set `blocked(`p`)` that contains the universes for which p could not immediately acquire or preempt. More formally:

$$\mathsf{Blocking}(p, u) \;\Leftrightarrow\; u \in \texttt{blocked(}p\texttt{)}$$

By simple substitution, we derive that the direct-waiting condition is true if and only if

$$\begin{aligned} & \texttt{empty(msgQ(}p\texttt{))} \\ \wedge\; & \texttt{pending(}p\texttt{)} > 0 \end{aligned}$$

$$\wedge \ \texttt{pending}(p) = \texttt{size}(\texttt{blocked}(p))$$

Thus, the restart condition can now be checked by testing that the above condition persists for a sufficiently long time.

6.3 Algorithms

We integrate our modified cautious-waiting heuristic into the deadlock-avoiding algorithms by refining the `getMessage` primitive. Figure 20 depicts the refinement, called `getMessageCooperatively`, which replaces `getMessage` in the overall system. In contrast to `getMessage`, `getMessageCooperatively` checks the direct-waiting predicate concurrently with waiting for message reception, and invokes a restart operation if predicate is satisfied. This concurrent check is handled by the alternate conditions of the **wait until** primitive. The first alternative is taken when there is a message in the message queue for process `p`, in which case the message is dequeued and returned in a fashion identical to the original `getMessage` function. The second alternative is taken only if the first alternative fails (per our definition of **wait until**). Consequently, we know that: (1) the message queue is empty and (2) `pending(p) > 0`.[1] This case checks the remaining conjunct (i.e., that the size of the blocked set equals the number of pending requests) and that the timeout has expired. If these conditions are true, then process `p` must be restarted.

The restart procedure must take care to avoid starvation. Consider, for example, that process p directly waits for process q, which directly waits for another process r due to

[1] because `getMessageCooperatively` is called from `nextAcquired` only if (1) is false or (2) is true.

```
function
    getMessageCooperatively( p : Process )
returns MessageType × Universe is
begin
    loop
        wait until (¬ empty(msgQ(p))) then
            LOCK(p);
            msg, u := dequeueMsg(msgQ(p));
            UNLOCK(p);
            return ⟨ msg, u ⟩;
        or until (size(blocked(p)) = pending(p)
                ∧ timeout(p)) then
            start_time(p) := current_time();
            dfsRestart(witness(p), p);
            increaseTimeout(p);
        end
    end
end

proc dfsRestart( u : Universe,
                 p : Process ) is
begin
    for (s ∈ accessibles(u)) do
        if (¬ marked(p,s) ∧ s ∉ Core(p)) then
            if( reorder(p,s) ) then
                dfsRestart(s, p);
            end;
        end;
    end;
end
```

```
function reorder( p : Process,
                  u : Universe )
returns boolean is
    var pOwnsU := false;
begin
    LOCK(u);
    mark(p,u);
    if (owner(u) = p) then
        acquisition_time(u) := start_time(p);
        pOwnsU := empty(reqQ(u));
        UNLOCK(u);
        if ( ¬ pOwnsU ) then
            surrender(u, p);
        end
    else
        reorderQueue(reqQ(u));
        UNLOCK(u);
    end;
    return pOwnsU;
end
```

Figure 20: Refinement of function `getMessage` to support deadlock recovery.

conflicting requests on a single universe u. Such a situation will trigger a restart, causing p to surrender and then attempt to re-acquire u. Moreover, after surrendering u, p will update its start time, thereby lowering its priority. Now assume that immediately after p surrenders u, but before p can update its start time, the (OS-level) scheduler gives both q and r enough time to successively acquire, release, and re-request u. Then when p is scheduled and tries to re-acquire u, it will once again find itself indirectly waiting on u relative to q, and this will trigger another restart operation. Theoretically, this can continue indefinitely, leading to starvation.

To avoid starvation, we manipulate the timeout interval T. Observe that if T is large enough, starvation cannot occur, because p will be able to acquire the universes released by r and q before an indirect waiting condition is asserted by the expiry of the timeout. Of course, if T is too large, then the time it takes to recover from a real deadlock will be at least T. To reconcile these concerns, T is initially set to a small value and increased after each restart. Eventually, T will be large enough so that in our example p will not restart before r and q have completed. At this point p becomes the highest-priority process and can preemptively acquire the universes under contention and complete its realm. A detailed discussion of this starvation-prevention mechanism can be found in [FP97], along with advice on how to select values for T.

Thus, restarting a process comprises three steps. First, the `start_time` of `p` is updated. Second, the procedure `dfsRestart` is invoked upon the witness of the last realm-affecting operation. Its purpose is to perform a depth-first search of $R_{\mathtt{p}}^{+}$, starting at `witness(p)`, updating the attributes of each universe to reflect the reordering of $\langle p, u \rangle$. Finally, the timeout of `p` is increased.

Function `dfsRestart` has two parameters: a universe `u` at which to start the traversal and the restarting process `p`. The for-loop and outermost if-condition implement a depth-first search with a marking strategy. Core universes are considered to be marked by default[2], and non-core universes are marked using the procedure `mark`, whose implementation we elide for brevity. When an unmarked universe *s* is traversed, we attempt to implement the **reorder**(*p*, *s*) transition, after which either *p* owns *s* or it does not. If *p* owns *s* after the reordering, we invoke the traversal recursively, as negotiations involving universes accessible from *s* will need to be reordered.

Function `reorder(p,u)` implements the transition **reorder**(*p*,*u*), returning true if *p* owns *u* after the transition. There are three cases to consider. First, suppose the owner of `u` is `p`. In this case, we have to update the acquisition time of `u` to reflect the reprioritization of `p`. If `u` is not under contention, (i.e., the request queue of `u` is empty), then there is nothing else to do to reorder $\langle p, u \rangle$, and `reorder` returns true to indicate that `p` (still) owns `u`. Otherwise, `u` it must be surrendered.[3] If `p` does not not own `u`, reordering $\langle p, u \rangle$ reduces to modifying the entry for `p` in the request queue of `u` to reflect to the new (lower) priority of `p`.

[2]Because restarting a process changes its start time, the test for a universe being in the core must refer to the initial start time assigned at the beginning of realm update. In our implementation, we provide a process attribute called `initial_start_time(p)`, which is consulted by functions `dfsRestart`, `surrender` and `wound`.

[3]Observe that the `surrender` operation checks independently that the universe is not in the core. Therefore, the acquisition time is updated in this case as well, so that `surrender` will not mistakenly believe that the universe is in the core.

Chapter 7

Feature: Conditional Contract Negotiation

We conclude our treatment of contract negotiation by refining the algorithms to handle conditional contracts. To motivate this treatment, observe that exclusion contracts are subsumed by conditional contracts in that an exclusion contract of the form u is semantically equivalent to a conditional contract of the form u **when** *true*. By analogy, a conditional contract with a (non-trivial) condition behaves exactly like an exclusion contract when the condition is verified. However, if the condition is not verified, then any migration that would have occurred as a result of negotiating the corresponding exclusion contract must be deferred until the condition can be verified. Negotiations that are deferred have much in common with negotiations that are blocking; thus we generalize our model of the negotiation of exclusion contracts to handle conditional contracts by refining what it means for a negotiation to be blocking.

Suppose p owns a universe v whose constraint contains a conditional universe reference

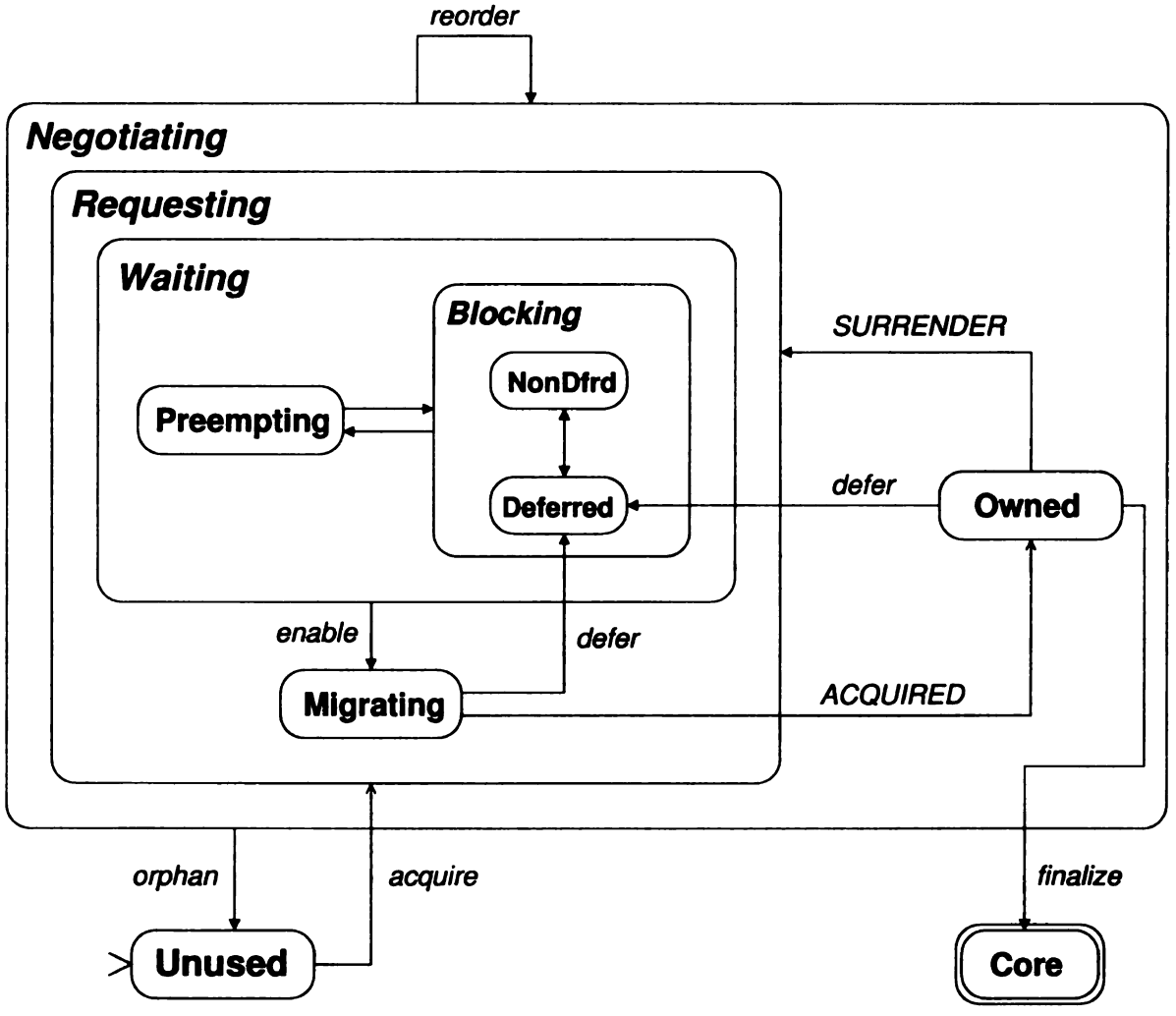

Figure 21: Refinement of protocol in Figure 19 to negotiate conditional contracts. This diagram depicts the most refined negotiation protocol.

of the form u **when** c for some (possibly trivial) condition c. Then, the negotiation $\langle p, u \rangle$ is *blocking* if

1. p lacks the priority to link the edge $v \rightarrow u$ into its realm; **or else**

2. u and v fail to verify c.

To say that p has priority to link u means either no other process currently owns u, or $u \notin Core(owner(u))$ and p is the oldest process contending for u. Observe that if c is the trivial condition, this definition reduces to the definition of a blocking negotiation for an exclusion contract, again as expected. A *deferred negotiation* is one that is blocking because of condition (2).

Figure 21 depicts the final (most refined) model of the negotiation state space. In this refinement, state Blocking partitions into substates Deferred and NonDfrd[1]. Briefly, a negotiation is in state NonDfrd if the blocking is caused by condition (1) above; whereas the negotiation is in state Deferred if the blocking is caused by condition (2). Observe that a negotiation over an exclusion contract can only block in state NonDfrd; whereas a negotiation over a (non-trivial) conditional contract can block in either NonDfrd or Deferred. Observe also that in a deferred negotiation, p will have checked condition c and found it not verified by u and the client of the contract; whereas in a non-deferred but blocking negotiation, p will not have checked c and will thus have no knowledge of its truth assignment.

Figure 21 also depicts two new transitions, both named defer, from Owned and Migrating into Deferred. These transitions arise when p negotiates for u to satisfy *multiple* contracts, one or more of which are exclusion contracts and one or more of which are con-

[1] short for non-deferred.

ditional. Consider the case where p, while negotiating an exclusion contract, has acquired (or is in the process of acquiring) u. If p then attempts to negotiate a non-trivial conditional contract whose referent universe is u, and if the u and the client fail to verify the referent condition, then p must release u, and $\langle p, u \rangle$ must enter **Deferred**. That is, unsatisfied conditional contracts trump exclusion contracts that refer to the same universe, for otherwise p would hold onto u but could not modify it to make the condition true.

To implement deferred negotiations and the transitions into and out of state **Deferred** requires associating with the negotiation any referent conditions of contracts that pertain to the universe under negotiation. Because these conditions are specific to a given negotiation, and not merely to a given universe, we store the set of them as part of the tuple that is placed in `reqQ(`u`)` (i.e., in addition the process identifier and reference count). When a process releases a universe u and looks to the request queue of u to discover which negotiation to enable, the process must first check to see if the relevant conditions are true. These additional checks involve straightforward additions to procedures `link`, `acquire`, `enterWaiting`, and `release`. Finally, observe that unavoidable deadlocks may arise due to deferred negotiations. However, because deferred is a substate of **Blocking**, the algorithms presented in Chapter 6 will handle those deadlocks.

Chapter 8

Eiffel Language Extensions for the Universe Model

We extended the Eiffel programming language with language constructs and libraries that support the universe model and implemented a compiler for these extensions. This chapter documents the language features available to a programmer who wishes to use this compiler. The core part of the language extension is the reification of universes (Section 8.1) and processes (Section 8.2) by means of language constructs and library classes. These language constructs and the supporting library classes allow the definition and instantiation of universes, processes, and their realms. Section 8.3 describes our handling of global data. Local data exists on the stack of a process and can always be accessed in a thread-safe fashion; heap data is partitioned into universes and access to it is regulated by the mechanisms of the universe model. Conversely, access to global data—which, by definition, is accessible to all processes—requires special precautions. Finally, the universe model requires special assignment semantics (Section 8.4). Under these semantics, it is sometimes

necessary to implement the assignment of an object reference to a variable by copying the referent in order to prevent inadvertent sharing of data between two processes. The chapter concludes with a summary of the new language features (Section 8.5).

8.1 Defining and Creating Universes

The reification of universes involves declaring universes via language constructs and the creation of runtime instances of universes. We declare universes by writing universe classes. Instantiating a universe class creates both an object, called a universe root, and a new universe, whose sole member is the universe root. In this section, we first describe how universe classes can be defined (explicitly, via inheritance, or by means of a type modifier) and then define the semantics of the universe creation process.

8.1.1 Defining Universes

A universe class is defined by prefixing a class definition with the keyword **universe**. Thus, the definition

universe class U . . . **end**

creates a universe class named `U`.

A universe class definition can also declare one or more concurrency constraints, introduced by the keyword **concurrency**. For example, the following class definition defines a universe class `U2` with a concurrency constraint <*constraint*>.

universe class U2

...

concurrency

<constraint>

end

Concurrency constraints follow the abstract syntax defined in Chapter 3. Operators have the standard Eiffel precedence and associativity [Mey92]. The new **when** operator is not associative and has a higher precedence level than the **and** operator and lower precedence than the comparison operators in standard Eiffel. The new => operator is synonymous with the existing **implies** operator and has the same precedence and associativity.

In addition to the concurrency constraint language defined in Chapter 3, our extension allows a convenient shorthand for arithmetic expressions. For example, consider the universe class definition in Figure 22, which provides a simple emulation for a low-level reentrant mutex mechanism. This class provides methods to lock and unlock the universe referenced by the variable `target_universe`. Calls to `lock` and `unlock` can be nested, and a `depth` variable ensures that the universe is released only if the same number of calls to `unlock` as to `lock` have occurred. The source code in the example is complicated, because the code must keep track of the nesting depth and update a boolean condition variable accordingly. Figure 23 shows a simpler implementation. Rather than having an explicit condition variable, the expression *depth* > 0 is used, greatly simplifying the code.

Generally, any legal Eiffel expression that uses only variables and constants of type `BOOLEAN`, `INTEGER`, `REAL`, and `DOUBLE`, arithmetic operators (including addition, sub-

```
 1   universe class REENTRANT_LOCK
 2   feature
 3       target_universe: UNIVERSE_BASE
 4       locked: BOOLEAN
 5       depth: INTEGER
 6   feature
 7       lock is
 8       do
 9           if depth = 0 then
10               locked := true
11           end
12           depth := depth + 1
13       end
14       unlock is
15       do
16           depth := depth - 1
17           if depth = 0 then
18               locked := false
19           end
20       end
21   concurrency
22       locked ⇒ target_universe
23   end -- class REENTRANT_LOCK
```

Figure 22: A reentrant mutex

```
1   universe class REENTRANT_LOCK
2   feature
3       target_universe: UNIVERSE_BASE
4       depth: INTEGER
5   feature
6       lock is
7       do
8           depth := depth + 1
9       end -- lock
10      unlock is
11      do
12          depth := depth - 1
13      end -- unlock
14  concurrency
15      (depth > 0) ⇒ target_universe
16  end -- class REENTRANT_LOCK
```

Figure 23: Improved reentrant mutex

traction, multiplication, division, and modulo), and comparison operators, can be used in lieu of a boolean condition variable. The compiler will treat each such expression as though there were a separately declared boolean condition variable that is updated each time the expression changes.

8.1.2 Universe Classes and Inheritance

Universe classes can be inherited by other universe classes. When one universe class (the child) inherits from another universe class (the parent), the child inherits the instance variables, methods, class invariants, and concurrency constraints defined in the parent. For example, a universe class can inherit from the class in Figure 22 and use the `lock` and `unlock` methods to acquire and release exclusive access to the `target_universe`

without having to redefine the concurrency constraint. In the case of multiple inheritance, the inherited constraint is the logical conjunction of the constraints of all parents. Like any other class, a parent universe class can also be *deferred*, which means that the class declares one or more abstract operations. (Such classes are often called abstract classes in other languages.)

If a universe class does not explicitly inherit from another universe class, then it implicitly inherits from the special class `UNIVERSE_BASE`, which is provided by our standard library. This predefined class provides the instance variables and methods necessary to implement the realm update algorithm (Chapter 4). Any universe class is either a direct or indirect descendant of `UNIVERSE_BASE`. Therefore, a variable that is defined to be of type `UNIVERSE_BASE` can hold a reference to any universe root in the program.

Inheritance provides a convenient way for the programmer to reuse concurrency and synchronization patterns (such as the one embodied in the `REENTRANT_LOCK` class). Thus, inheritance allows a software designer to easily reuse predefined client-side synchronization logic.

8.1.3 Alternate Universe Declaration

It is sometimes inconvenient to define a new class to declare a universe. For example, if a universe class contains no concurrency constraint, then its definition differs from that of a normal class only by the presence of the keyword **universe**. As a convenience feature, our language extension provides a shorthand notation for such simple universe classes.

Given an existing type T, the type declaration **universe** T declares a universe class with

the same class signature (i.e., methods, instance variables, and class invariants) as *T*. For example, the declaration

feature

x: **universe** HASH_TABLE;

declares an instance variable `x` of a universe class that has the same class signature as `HASH_TABLE` and the concurrency constraint `true`. As with any other universe class without an explicit inheritance clause, this class also implicitly inherits from `UNIVERSE_BASE`.

8.1.4 Creating Universes

Universe classes are instantiated in the same fashion as any other Eiffel class, using either the **create** instruction or the alternate `!!` syntax. Given an instance variable

feature

x: **universe** T;

where `T` has a constructor `make` with no arguments, each of the following instructions create a universe.

create x.make

!! x.make

After either instruction has completed, `x` will contain a reference to the universe root of the newly created universe.

Observe that for the constructor (i.e., `make`) to be called, the newly created universe must be part of the realm of the process that executes the **create** instruction. Otherwise, the process would be accessing an object outside its realm. The language implementation must therefore adjust the realm of the current process to include the newly created universe. This adjustment occurs after the universe and the universe root have been created, but before the constructor is called. Similarly, after the creation instruction has been completed, the universe must again be excluded from the realm. Otherwise, it might be impossible for other processes to acquire the newly created universe.

More precisely, we adjust the realm both upon entry into and upon exit from the constructor. Upon entry, the newly created universe is pushed on the universe stack (cf. Section 3.4) and the realm update algorithm is invoked to add the universe to the realm. The constructor is then executed. Upon return from the constructor, the universe is removed from the universe stack and the realm update algorithm is invoked again to remove the universe from the realm.

8.2 Defining and Creating Processes

Like universes, processes are reified by declaring them via language constructs and then creating run-time instances. The declaration must provide the code that the process will execute. The runtime instantiation consists of both a representation of a process[1] in the underlying operating system and a realm within which the process will execute.

[1] As before, we include light-weight processes and threads in the notion of operating system processes.

8.2.1 The Constituent Parts of a Process

To define the code that a process will execute, the programmer must create a class that inherits from the predefined class PROCESS_BASE (an excerpt of which is presented in Figure 24) and that defines a `start` method, which contains the code that the process will execute. For example, Figure 26 defines a class EVENT_MONITOR. A process executing the `start` method defined in EVENT_MONITOR will enter an infinite loop, constantly retrieving events from an event channel and handling them, but never terminating.

Creating a running process from such a descendant of PROCESS_BASE comprises two steps, the creation of an initial universe to populate the realm of the process, and the creation of an operating system process that executes the code provided by the `start` method. The first step is performed by creating an instance of the descendant of PROCESS_BASE. This instance will be the universe root of the initial universe with which the realm of the process will be populated. We call it the *process root* of the process.

Second, a process in the underlying operating system must be created that will then execute the `start` method of the process root. This step is performed by creating an instance of the standard library class PROCESS, which represents operating system processes. (An excerpt of the class can be seen in Figure 25.) This class has a constructor called `with_root`, whose sole argument is the process root of the new process. Invoking the constructor will create an operating system process that will populate its realm with the supplied process root and will then execute the root's `start` method.

The precise semantics of the `with_root` constructor are as follows. Given a variable `process` of type PROCESS and another instance `process_root` of a universe class

```
1    deferred universe class PROCESS_BASE
2    feature
3        start is deferred end
4    end -- class PROCESS_BASE
```

Figure 24: The PROCESS_BASE class

that inherits from PROCESS_BASE, the instruction

create process.with_root(process_root)

creates a new operating system process. This new process then performs the following operations. First, it pushes the universe referenced by `process_root` on the universe stack and invokes the realm update algorithm. The result of the realm update will be that the process root is inserted into the realm of the new process. Second, the new process calls `process_root.start`. This operation is safe, because `process_root` is now part of the realm of the new process. Third, once the `start` method terminates (either normally or by an uncaught exception), the process releases all universes in its realm and terminates itself. Note that the original process (i.e., the one that executed the **create** instruction) will not block while the new process performs these steps, but will continue concurrently. The result of the **create** instruction in the original process will be that `process` now references an instance of PROCESS that provides a handle for the new process. Thereafter, `process` can be used to manipulate the attributes of the new operating system process (such as its priority).

Creating processes in two independent steps is necessary to gain maximum control over the details of process creation. In particular, the two-step process allows the programmer to obtain a reference to both the process root of the process and the operating system rep-

```
1   class PROCESS
2   creation with_root
3   feature { NONE }
4       with_root(process_root: PROCESS_BASE) is
5       do
6           ...
7       end
8   end -- class PROCESS
```

Figure 25: The PROCESS class

resentation of the process. Otherwise, if the creation of a process root (i.e., an instance of a descendant of PROCESS_BASE) implicitly also created a new operating system process, it would not be possible for the programmer to reliably identify or otherwise reference that process, because the only output of the constructor of the process root is the process root itself. If the flexibility of the two-step process is not needed, the programmer can design simpler custom procedures to create processes.

8.2.2 An Example of Process Creation

For a non-trivial example of process creation, consider Figure 28, which uses the EVENT_MONITOR class from Figure 26 and the EVENT_GENERATOR class from Figure 27. The EVENT_GENERATOR class generates events and inserts them into an event channel. At the other end of the event channel, an instance of EVENT_MONITOR retrieves the generated events from the channel and processes them.

Initially (lines 12–14), instances of the event channel, monitor, and generator are created. Each instance is a universe root, and `channel` is passed as an argument to the constructors of the monitor and generator universe classes, so that they can reference it. Once

```
1   universe class EVENT_MONITOR
2       inherit PROCESS_BASE
3   creation make
4   feature
5       event_channel: EVENT_CHANNEL
6       reading: BOOLEAN
7       make(channel: EVENT_CHANNEL) is
8       do event_channel := channel end
9       start is
10      local event: EVENT
11      do
12          from until false loop -- infinite loop
13              reading := true
14              event := event_channel.dequeue
15              reading := false
16              handle(event)
17          end
18      end
19  concurrency
20      reading ⇒ event_channel when event_channel.size > 0
21  end -- class EVENT_MONITOR
```

Figure 26: An event monitor

```
1   universe class EVENT_GENERATOR
2       inherit PROCESS_BASE
3   feature
4       event_channel: EVENT_CHANNEL
5       writing: BOOLEAN
6       make(channel: EVENT_CHANNEL) is
7       do event_channel := channel end
8       start is
9       do ... end
10  concurrency
11      writing ⇒ event_channel
12  end -- class EVENT_GENERATOR
```

Figure 27: An event generator

```
1   universe class MAIN
2       inherit PROCESS_BASE
3   feature
4       start is
5       local
6           generator: EVENT_GENERATOR
7           monitor: EVENT_MONITOR
8           channel: EVENT_CHANNEL
9           monitor_process, generator_process: PROCESS
10      do
11          -- create universes
12          create channel.make
13          create monitor.make(channel)
14          create generator.make(channel)
15          -- create processes
16          create monitor_process.with_root(monitor)
17          create generator_process.with_root(generator)
18          -- suspend self
19          suspend
20      end
21  end
```

Figure 28: Creating processes

the universes have been created, the next step is to create and start the processes (lines 16–17). First, the monitor process is created by instantiating `PROCESS` with the constructor `with_root`. The argument of the constructor is the root of the monitor universe. The `with_root` constructor builds the initial realm for the monitor process so that it includes the monitor universe and then calls `monitor.start`. The same procedure is then repeated for the generator process and the generator universe. Finally, the main process suspends itself via the standard library method `suspend` (line 19) so as not to take CPU time away from the generator and monitor processes.

The duration of the `start` method determines process lifetime. Just as a process starts by beginning to execute this method, a process terminates when the `start` method of its process root returns, either normally, or via an exception. In our example, the `start` method of the monitor process contains an infinite loop. Because the monitor process will never return from `start`, it does not terminate, but runs indefinitely.

Figure 28 also shows how to define the initial process of the system. A specially designated class `MAIN`[2] describes the code and the initial realm of the initial process of the system. The runtime system will execute this process implicitly at program start, and as soon as the initial process terminates, the entire program terminates as well. Therefore, we use the `suspend` method to keep the program active[3]. Alternatively, the `MAIN` class could subsume the functionality of either the event monitor or the event generator process, thereby eliminating the need for that process. This design would obviate the call to `suspend`.

---

[2]The name of this class can be changed by setting a compiler option.

[3]Without the call to `suspend`, the program would immediately terminate.

In summary, the reification of a process requires that the programmer define a descendant of PROCESS_BASE and instantiate both that descendant and the standard library class PROCESS. The descendant provides the code for the process to execute; the instance of the descendant populates the initial realm of the process; and the instance of PROCESS reifies the underlying operating system process that executes the code of the descendant.

8.3 Accessing Global Data

Like all programming languages, Eiffel allows access to global data (as opposed to the local data of methods and the heap data that makes up the instance variables of an object). Such access can be regulated in one of two ways. First, each item of global data (which may be a single global variable or the variables of an entire module) can be treated as a universe. A reference to such data can then be assigned to the universe variable of an object and access to the data can be regulated by means of concurrency constraints. As an alternative, storage for global data can be replicated for each process. Then each process would have its own separate instance of the global data. For example, a global variable could be replicated for each process, and each process could access and alter its own version independently without affecting the others. The second approach makes each replicated version of the storage global only for a single process (and such data is therefore often called thread-local data), whereas in the first approach, data is globally shared by the entire program (at the expense of having to explicitly regulate access to it).

In Eiffel, global data is accessed by means of so-called *once functions*. The syntax of a once function differs from that of a regular function by the use of the keyword **once** instead

of **do** to denote the start of the function body (cf. Figure 29, line 4). A once function in standard Eiffel is evaluated only once per program run: Even if it is called several times, the result is calculated only the first time, and stored in a global location. On any subsequent call, the result that was calculated the first time is retrieved from that location and returned. In this respect, once functions implement the Singleton design pattern [GHJV95]. Note that in a concurrent system, initializing and accessing the global location must occur in a thread-safe fashion.

In the universe model, the semantics of a once function depend on whether the return type of the function is a universe class or not. If the return type is a universe class, then the once function is evaluated once per program and access to the resulting universe is regulated normally via concurrency constraints. Otherwise, the once function is evaluated *once per process* and the result is stored in thread-local storage of the process, where it can be accessed without interference by other processes. In the sequel, we illustrate both cases.

Figure 29 shows an alternate implementation of the event channel example that uses a once function whose return type is a universe class to represent the globally shared event channels. The class `EVENT_GLOBALS` declares a single once function (`global_channel`, lines 3–6). Its body creates an instance of `EVENT_CHANNEL`. `Result` is a special variable, which holds the return value of a function and is of the same type as the function itself. It can be used like any other variable: It can be the target and source of assignments and the target of a creation instruction. Therefore, the result of `global_channel` will be a globally unique instance of `EVENT_CHANNEL`, no matter how often the function is called.

Other classes can now use the function defined in `EVENT_GLOBALS` by inheriting

from EVENT_GLOBALS. The property that a once function will be called only once per program remains true even when it is inherited: When two different classes inherit from EVENT_GLOBALS, both will obtain the same result when calling the inherited function global_channel. In our example, both EVENT_GENERATOR and EVENT_MONITOR can therefore reference a shared event channel by accessing the global_channel function. Lines 13–16 of Figure 29 demonstrate how a constructor has to be written to use that function. The class definition of EVENT_MONITOR is very similar and has been elided for brevity.

The start method of the program can now be written without reference to the channel universe (lines 22-36). The creation of the channel universe has been omitted entirely, since it will be automatically created the first time that global_channel is called. Lines 29–30 create the monitor and generator universes, but do not pass an argument to their constructors.

Figure 30 illustrates the case where the return type of a once function is not a universe class. Here, primes is part of a hash table implementation. However, by the semantics of standard Eiffel (that primes is evaluated only once per program), the result would be shared among all processes. This could violate the basic principle that only one process may access an object at any time.

To correct the behavior for functions whose type is not a universe class, we require that a once function is evaluated once per program execution only if the type of the result of the function is a universe class. If the type of the result of the function is not a universe class, then the function is evaluated *once per process*. The result of the function is stored in thread-local storage for each process. Therefore, no two processes can access the result

```
1   universe class EVENT_GLOBALS is
2   feature
3       global_channel: EVENT_CHANNEL is
4       once
5           create Result.make
6       end -- global_channel
7   end -- class EVENT_GLOBALS
8   universe class EVENT_GENERATOR
9       inherit PROCESS_BASE; EVENT_GLOBALS
10  creation make
11  feature
12      event_channel: EVENT_CHANNEL
13      make is
14      do
15          event_channel := global_channel
16      end
17      ...
18  end -- class EVENT_GENERATOR
19  universe class MAIN
20      inherit PROCESS_BASE
21  feature
22      start is
23      local
24          generator: EVENT_GENERATOR
25          monitor: EVENT_MONITOR
26          monitor_process, generator_process: PROCESS
27      do
28          -- create universes
29          create monitor.make
30          create generator.make
31          -- create processes
32          create monitor_process.with_root(monitor)
33          create generator_process.with_root(generator)
34          -- suspend self
35          suspend
36      end
37  end
```

Figure 29: Accessing global data

```
1   class HASH_TABLE
2   feature
3       primes: ARRAY[INTEGER] is
4       once
5           -- calculate primes in Result
6       end
7       ...
8   end -- class HASH_TABLE
```

Figure 30: Thread-local once functions

of the once function at the same time.

Once functions therefore provide convenient access to both global and thread-local data. Global data is defined as a once function that returns an instance of a universe class. In order to access global data defined through a once function, the result of that function has to be assigned to a universe variable, and a concurrency constraint must exist that makes the universe accessible. Conversely, thread-local data is defined as a once function that returns an instance of a normal class and can be accessed safely at any time without the need to manipulate concurrency constraints. [CW02] derived similar semantics for once functions for an implementation of Eiffel's SCOOP mechanism that were omitted from the original specification [Mey93, Mey97].

8.4 Assignments and Data Access

The condition imposed by the universe model that no process access objects outside its realm requires adjustments to the semantics of assignments and method invocations. In particular, assignments of references to variables and the passing of references as actual

parameters to methods can result in more than one process referencing the same object. In order to avoid uncontrolled concurrent access to such objects, one has to either guard each access with a runtime check that ensures that accessed objects are in the realm of the current process, or replicate objects so that they cannot be referenced by more than one process. In the sequel, we show how these requirements affect the semantics of assignments and method invocations.

8.4.1 Motivation

As a motivating example, consider line 14 of Figure 26. This assignment retrieves a reference to an instance of `EVENT` from the `event_channel` universe and assigns it to the `event` variable. If the data referenced by `event` were part of the channel universe, then once `reading` had been set to `false` (in line 15), the `process` method (that is called in line 16) would be unable to legally access the object referenced by `event`.

To avoid this problem, we treat any assignment where the source might be in a different universe than the destination differently from normal assignments: A copy of the source object is created and assigned to the destination variable. Thus, the `process` method can safely work on `event` (which has been copied into the event monitor universe) without crossing realm boundaries.

However, creating such a copy is potentially an expensive operation. Not only does the object itself have to be copied, but, if any of its instance variables reference other objects, those objects may have to be copied as well, proceeding recursively through all objects that are reachable directly or indirectly from the source of the assignment. Therefore, while

simple data such as an integer or string is handled efficiently by the copying approach, it can be very expensive in general. It is therefore desirable for the programmer to be able to suppress such copying when it is not warranted.

8.4.2 Shared and Normal References

We alter the semantics of assignments in our language extension to accommodate these requirements. The normal semantics of assignment prescribe that assigning a reference to a variable (a *reference assignment* for short) will not create a copy of the referent. Instead, the semantics of reference assignment are preserved only when it can be guaranteed at compile time that the destination and source of the assignment lie within the same universe. Observe that for a strictly sequential program—a single process existing in a single universe—the semantics of reference assignment will not be affected at all. Thus, the alteration of reference assignment semantics only affects newly written concurrent code (where the programmer is aware of these semantics), but will leave existing sequential code intact.

In addition to altering the semantics of normal references, we also provide *shared references* that allow reference assignments without creating a copy, even if the source and the destination of the assignment do not lie within the same universe. Given a type T, the declaration of a variable as being of type **shared** T will mark the variable as a shared reference. (We call such a variable a *shared variable* for short.) As with the **universe** keyword, a class can also be declared as shared. For example, the class definition

shared class S ... **end**

declares a shared class `S`, which has the same class signature as a class definition without

the **shared** prefix. A variable declaration such as:

feature

x: S

then makes the declared variable x a shared variable. A shared variable can hold references to objects in any universe, whereas normal variables can only hold references to objects in the same universe. Note that variables whose type is a universe class behave like shared variables in that they can hold a reference to any universe root, regardless of the universe of which the referent is a root. Therefore, assigning an object reference to a shared variable or to a variable whose type is a universe class will not create a copy of the referent, even if the referent is contained in a different universe. It is the responsibility of the programmer to make sure that concurrency constraints allow access to the objects referenced by shared variables.

To illustrate the use of **shared**, Figure 31 shows a variant of the event monitor code from Figure 26 that uses a shared variable. The main difference is the use of the keyword **shared** (line 11). Because `event` is now shared, the call of `handle(event)` (line 16) will access data contained in the event channel. Therefore, the realm-affecting assignment to `reading` now has to occur after the call of `handle(event)`, because otherwise the event channel universe would be removed from the realm and `handle` could not access it.

Because a shared variable can point to any object in the system, each access to a shared variable must be preceded by a runtime check (which is generated by the compiler). That check verifies that the referent is part of a universe that is in the realm of the current process. Should the check fail, a runtime exception is raised. The same check is necessary when

```
1   universe class EVENT_MONITOR
2       inherit PROCESS_BASE
3   creation make
4   feature
5       event_channel: EVENT_CHANNEL
6       reading: BOOLEAN
7       make(channel: EVENT_CHANNEL) is
8       do event_channel := channel end
9       start is
10      local
11          event: shared EVENT
12      do
13          from until false loop -- infinite loop
14              reading := true
15              event := event_channel.dequeue
16              handle(event)
17              reading := false
18          end
19      end
20  concurrency
21      reading ⇒ event_channel when event_channel.size > 0
22  end -- class EVENT_MONITOR
```

Figure 31: Shared Data

accessing an instance of a universe class. A variable that is declared to be of a universe class can, like a shared variable, reference objects in any universe. Conversely, normal references only reference objects in the same universe and do not require a check.

8.4.3 Locality of Objects and Variables

The semantics of a reference assignment in our language extension depend on the location of the source and the destination of the assignment. In order to make these semantics precise, we define what it means for an object or variable to be *local to a universe*. An object is local to a universe if and only if it is contained within that universe. An instance variable

of an object is local to a universe if and only if the object is contained within that universe. A local variable of a method (including its formal parameters and the special *Result* variable) is local to a universe if the current object of that method (i.e., the value of *Current*) is contained within that universe. Note that any variable or object is local to exactly one universe, which we call the *containing universe* of the variable or object.

We also define whether a type is local or not. A type is *local* if and only if any variable of that type can only reference or contain objects whose containing universe is the same as the containing universe of the variable. Otherwise, we say the type is *non-local*. In Eiffel, both shared and universe classes are non-local types. All other classes are local types.

8.4.4 Semantics of Assignment

Whether the source of an assignment has to be copied before being assigned to a variable depends on the source and the destination of the assignment. Semantics vary depending on whether the source and destination are local to the same universe. To generate efficient code for assignments, we infer this information at compile time from the types of the source and destination of the assignment.

An assignment *var* := *expr* in Eiffel has only a limited number of variants, which we will enumerate in the sequel along with their semantics. The destination *var* of the assignment can only be an instance variable or the local variable of a method (local variables include the formal parameters of the method and the special *Result* variable that holds the result of a function). The source *expr* can be either a local variable, an instance variable, or a method call. Note that operators in Eiffel are just an alternative syntax for a method call.

For instance, the expression $1 + 2$ is rendered as an invocation of the method `infix "+"` on the target 1 with argument 2. Similarly, constants can be interpreted as parameterless once functions, and access to instance variables of an object other than *Current* can be be implemented through parameterless methods that return the contents of the variable.

We consider first the case where both *var* and *expr* are local or instance variables. If the type of *var* is non-local, no copying is necessary, because a variable of a non-local type can reference an object in any universe. Likewise, if both *var* and *expr* are of a local type, then no copying is necessary, because *expr* can only reference objects in the same universe, and *var* can reference objects in the same universe. However, if *expr* is of a non-local type, and *var* is of a local type, then the assignment must assign a copy of *expr* to *var*.

The semantics of assignment where *expr* is a method call of the form

$$target.method(arg_1, \ldots, arg_n)$$

are more complex, because *target* may be either local to the current universe or not, which affects the containing universe of the result of the function. First, if *target* is declared to be of a local type, then the semantics of the assignment are the same as if *expr* were an instance variable of the current object; i.e., copying is necessary if and only if the type of the result of *method* is of a non-local type and *var* is of a local type. This is because the *Result* variable of a function that is declared to be of a local type can only reference an object local to the same universe that *var* is local to. Second, if *target* is of a non-local type, then we must assume that it is possible that the result of *method* is not local to the containing universe of *var*. While the method itself may have been declared to be of a

local type, the result will be local to the containing universe of *target*, not the containing universe of the object invoking the method. Therefore, the result of such a method call must be copied.

8.4.5 Semantics of Argument Passing

Assignment also implicitly occurs when a reference is passed as an argument to a method. Such argument passing will assign the reference to a formal parameter of the method. Without loss of generality, we consider only methods with a single argument to derive the semantics of such implicit assignments.

Consider a method call

target.method(*expr*)

where *expr* is assigned to a formal parameter *par* of *method* and *target* is declared to be of a local type. In this case, *par* is local to the containing universe of the current object and the semantics are identical to that of the assignment *par* := *expr*, where *par* is a local variable.

However, if *target* is declared to be of a non-local type, then the containing universe of *par* is the same as that of *target*, and *par* may not be local to the containing universe of *expr*. Under these circumstances, if *par* is declared to be of a local type, then the assignment has to create a copy of *expr*, which is then assigned to *par*. Conversely, if *par* is declared to be of a non-local type, then it can hold a reference to an object in any universe, and no copying occurs.

8.4.6 Assignment of Values

Eiffel's type system supports not only reference types, but also so-called *expanded* types. A variable that is declared to be of an expanded type does not contain a reference to an object, but instead contains the object itself. An expanded type in Eiffel would generally correspond to a record type in Pascal or Modula-2 or a struct type in C/C++, whereas a reference type would correspond to a pointer to a record or struct. Examples of expanded types include the basic arithmetic types, such as `INTEGER` or `REAL`, but also user-defined classes that are prefixed by the keyword **expanded**. The following class definition defines a class `COMPLEX` that is an expanded type.

```
expanded class COMPLEX
    real: DOUBLE
    imaginary: DOUBLE
    ...
end
```

In standard Eiffel, assignment between expanded types is performed componentwise. For example, if one has two variables `z1`, `z2` of type `COMPLEX`, then the assignment `z1 := z2` is performed by assigning `z2.real` to `z1.real` and `z2.imaginary` to `z1.imaginary`.

In our extension of Eiffel, assignment of values of expanded types also occurs componentwise. That is, the assignment of an expression *expr* to a variable *var*, where *var* and *expr* are of an expanded type, is replaced by an (atomic) sequence of assignments

$$var.x_1 := expr.x_1$$

...

$var.x_k := expr.x_k$

for each component x_i of the type of *var*. Because *expr* may have side effects, it is evaluated only once for each such sequence. If an x_i is itself of an expanded type, the process is repeated for that x_i until the original assignment has been replaced by a sequence of assignments of either reference types or expanded types that do not contain references (such as `INTEGER`). The resulting sequence of assignments then follows the basic rules for reference assignments described above.

8.4.7 Deep Copying

If an assignment to a variable of a local type necessitates copying by the above rules, this is a deep or recursive copy, which copies not only the object itself, but also objects referenced by instance variables of that object, and so forth. In this section, we define the precise semantics of the deep copy of a data structure.

The purpose of creating a deep copy is to avoid inadvertent access to an object that is not part of the realm of the current process. Consider, for instance, that the type `EVENT` in our example is defined as:

class EVENT

feature

priority: INTEGER

message: STRING

end

Copying only the event object itself from the event channel to the event monitor universe would leave the message object inside the event channel universe. If `process` were defined as

class EVENT_MONITOR

...

feature

process(event: EVENT) **is**

do

print(event.message)

end

...

end

it would access an object (`event.message`) that is not part of the current realm. The result of such an operation would be undefined. Note that if `message` were to be declared as being of type **shared** STRING, then the access would result in defined behavior, namely a runtime error if `message` is not part of the current realm. A deep copy operation transfers not only the object referenced by the source of an assignment to another universe, but also any object referenced directly or indirectly by instance variables (excluding those instance variables that are of a non-local type).

We first define what objects need to be copied. For this purpose, we define a *local reachability* relation on objects. An object o_2 is *locally reachable in one step* from an

object o_1 if o_1 has an instance variable of a local type that references or contains o_2. An object o_n is *locally reachable* from an object o_1 if there are objects $o_2, \ldots, o_{n-1}$ such that o_{i+1} is locally reachable in one step from o_i for all $i < n$, or if $o_1 = o_n$. Effectively, local reachability is the reflexive, transitive closure of one-step local reachability. The set of objects that are locally reachable from o_1 can be computed by recursively traversing all instance variables that are declared to be of a local type. Instance variables that are shared or whose type is a universe class are not traversed. Thus, the deep copying procedure will not access objects outside the realm of the current process.

Let S be the set of objects locally reachable from an object *source*. For all $o \in S$, we create a new object $copy(o)$ such that $copy(o)$ is an identical copy of o. We now have a copy of each object locally reachable from *source*. However, the instance variables of these copies may be incorrect. For instance, if object *source* had an instance variable (of a local type) that referenced *ob*, the corresponding instance variable of $copy(source)$ would still reference the original *ob* instead of $copy(ob)$. Therefore, these instance variables must be updated in a final pass.

For all $o \in S$, we update the instance variables of $copy(o)$. Let v be an instance variable of $copy(o)$. If v is declared to be of a non-local type, then it is not changed. If v is declared to be of a local type and it references an object $o' \in S$, then v is assigned a reference to $copy(o')$. If v is declared to be of a local type and it references an object $o' \notin S$, then v is not changed. Finally, if v does not reference an object (e.g., if it is of type `INTEGER` or another expanded type that does not contain references), then v is not changed, either. After all instance variables have been updated, $copy(source)$ contains a deep copy of *source*.

To illustrate this process, consider the instance diagrams in Figure 32. Each box in

this diagram describes an object, with the shaded rectangle at its top describing the name and type of the object and the white rectangles describing the instance variables and their types. An arrow emanating from a white rectangle points to the object that the corresponding instance variable references. The objective of this example is to show the process of a deep copy with *source* being the object $ob1$. The set S of objects reachable from $ob1$ are $\{ob1, ob2, ob3\}$. The object $ob4$ is not reachable, since it is only referenced by the shared variable $b2$ of $ob2$. The copying process then creates objects $ob1copy = copy(ob2)$, $ob2copy = copy(ob2)$, and $ob3copy = copy(ob3)$. Updating the instance variables of $ob1copy$ changes a to reference $ob2copy$ instead of $ob2$ and leaves b unchanged (since it is of a non-local type). Updating the instance variables of $ob2copy$ changes c to reference $ob1copy$ instead of $ob1$ and b to reference $ob3copy$ instead of $ob3$, while $b2$, being of a non-local type, remains unchanged. Finally, updating the instance variables of $ob3$ is not necessary, since its only instance variable is an integer which does not reference any object.

In summary, this definition of deep copying satisfies the requirement that objects outside the realm of the current process will not be accessed. The copying algorithm itself will only access objects in the same universe as *source*, which is already part of that realm. Once the data structure originating at *source* has been copied, it is not possible for user-defined code to access objects outside the realm via *copy*(*source*), either. All objects that can be accessed via *copy*(*source*) are either local to the same universe as *copy*(*source*) itself or are referenced by a variable of a non-local type and all accesses through variables of a non-local type are checked at runtime.

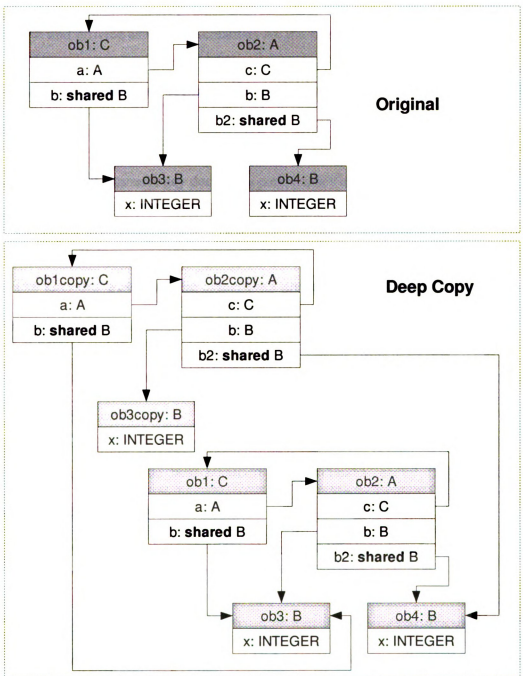

Figure 32: Deep Copy Example

8.5 Summary of Language Extensions

Our extensions introduce the new keywords **universe**, **concurrency**, and **shared**, the new symbol => and provide additional semantics for the existing keywords **when** and **once**.

- The keyword **universe** begins the declaration of a universe class (Section 8.1.1 and Section 8.1.3).
- The keyword **concurrency** declares a concurrency constraint (Section 8.1.1).
- The keyword **shared** declares a shared class or variable (Section 8.4.2).
- The keyword **when** and the symbol => are operators for concurrency constraints (Section 8.1.1)
- The keyword **once** declares once functions, which have special semantics in our extension (Section 8.3).

In addition, we provide new standard library classes, called `UNIVERSE_BASE`, `PROCESS_BASE`, and `PROCESS`.

- `UNIVERSE_BASE` is an ancestor for all universe classes (Section 8.1.2).
- `PROCESS_BASE` is a deferred class, from which other classes inherit to define the code of a process. This descendant is then instantiated to define the process root (initial universe) of a realm (Section 8.2.1).
- The instances of `PROCESS` are representations of actual processes as provided by the underlying operating system (Section 8.2.1).

Finally, we alter the semantics of assignment and method invocation to prevent inadvertent concurrent access to shared data (Section 8.4).

Chapter 9

Performance Evaluation

We now turn to the issue of how well the realm update algorithm performs. The performance of each of the individual components of our algorithm (e.g., wound-wait, cautious waiting, and reference counting) has been analyzed before in isolation [BKH97, HZ92, BAR+03]. Given that our algorithms combine these disparate techniques in a manner that involves complex interactions, a full complexity analysis is beyond the scope of this thesis. Instead, we chose to evaluate the algorithm empirically. To this end, we have devised a parameterized benchmark, which, dependent on a set of input parameters describing realm size and degree of contention, will yield the overhead of adding universes to a realm and removing them from it. We first introduce the benchmark in Section 9.1, including the parameters and their meaning. Section 9.2 explores the results we obtain for various settings of these parameters. Section 9.3 discusses the cost of adding universes to a realm or removing them from a realm on a uniprocessor machine. This cost is essentially linear in terms of the number of universes involved, with a small cost per universe, even under artificially high levels of contention. On a symmetric multi-processor architecture, execution cost for

our benchmarks is dominated by an OS-specific context switch overhead that is inherent to any completely fair scheduling approach.

9.1 The Benchmark Infrastructure

The benchmarks described in this section are generated from a simple class schema (Figure 33) and a set of input parameters. The generated code is passed to our extended Eiffel compiler, and the compiled code is then executed to estimate the overhead caused by the realm update algorithm for the given input parameters.

9.1.1 The Class Schema

The class schema (Figure 33) comprises a condition variable c (line 8), a set of universe variables $u_1, \ldots, u_n$ (line 9)[1], a concurrency constraint $c \Rightarrow u_1 \wedge \ldots \wedge u_n$ (line 30)[1] over these variables. The schema also provides a procedure `iterate` that repeatedly flips c, thereby implicitly invoking the realm update algorithm to either add (when c becomes true) or remove (when c becomes false) universes in the set $\{u_1, ..., u_n\}$. The body of this procedure consists of a pair of nested loops, the outer loop of which loops forever over its body. The body consists of three statements, the first of which is the inner loop (lines 18–20) that controls the interval between invocations of the realm update algorithm. Its duration is determined by the function `interval_length` (see below). The second is an instruction (line 21) to toggle the condition variable c. Changing c is a realm-affecting operation,

[1] Note that the code listed in the schema is a placeholder for the actual code. The code generator instantiates this schema to produce syntactically correct code.

and thus causes the invocation of the realm update algorithm. Finally, at the end of each iteration of the outer loop, the routine `record_iteration` is called, which is a special benchmarking routine (implemented in C and assembly language to avoid distortions of the benchmark results). Its purpose is to check whether a given number of iterations of the outer loop has been reached, and to end the benchmarking process once that happens. It does this by incrementing a global counter (shared by all threads) atomically, and if a set number of iterations is reached, it terminates the program and outputs the time spent since its inception.

The function `interval_length`, which determines the duration of the inner loop, and thus the time between invocations of the realm update algorithm, generates pseudo-random numbers with a fixed average. The purpose of having it generate pseudo-random numbers is to avoid having the realm update algorithm accidentally operate in lockstep with the operating system scheduler or other operating system events. The function is calibrated so that an average iteration of the outer loop takes ~ 1000 clock cycles. The pseudo-random numbers follow the standard exponential distribution over the interval $(0, \infty)$.

Observe that n may be zero, in which case we omit any declaration of universes in line 9, and set the concurrency constraint (line 30) to be $c \Rightarrow \mathit{true}$. Even though toggling the value in c has no effect on the satisfiability of this constraint, assignments to c are nevertheless realm affecting operations. Thus, they invoke the realm-update algorithm and allow us to estimate the overhead for the basic invocation.

9.1.2 Input Parameters

Three input parameters affect the generated benchmark code. The first parameter, *Number-Processes*, is a positive integer that sets the number of processes. Second, *DeltaRealm* is a non-negative integer that controls how many universes are added to (or removed from) the realm of a process on each iteration. *DeltaRealm* corresponds to the value of n above. Third, *Overlap* is a non-negative integer less than or equal to *DeltaRealm*, which defines the overlap between two processes: Suppose that processes are numbered $p_1, ..., p_k$. Then the maximum realm of p_i will have *Overlap* universes in common with the maximum realm of p_{i+1} for $1 \leq i < k$, and the maximum realm of p_k will have *Overlap* universes in common with the maximum realm of p_1.

9.1.3 Output Values and Precision

For any set of input parameters, we track two output values. One is the average *Overhead* incurred by invoking the realm update algorithm. We determine that value by running the benchmark twice, once with the realm update algorithm enabled, and once with the algorithm disabled. We disable realm updates by removing the invocation of the procedure from the code generated by the compiler for the `c := not c` instruction[2]. The difference between the two runs, divided by the number of total iterations performed, is the average overhead caused by the realm update algorithm per iteration. To minimize errors, we further average the result over several runs.

The other output value is *SignalRate*, the number of times per thousand invocations (‰)

[2]Observe that this is safe and does not generate any runtime errors, since no process actually ever accesses any of the u_i.

```
1   universe class BENCHMARK
2       inherit PROCESS_BASE
3
4   ...
5
6   feature { NONE } -- attributes
7
8       c: BOOLEAN
9       u1, ..., un: UNIVERSE_BASE -- declaration will be generated for n >= 0
10
11  feature { ANY } -- public routines
12
13      iterate is
14      local
15          j: INTEGER
16      do
17          from until false loop -- infinite loop
18              from j := interval_length until j = 0 loop
19                  j := j - 1
20              end
21              c := not c -- invokes scheduler
22              record_iteration
23          end
24      end
25
26  ...
27
28  concurrency
29
30      c => u1 and ... and un -- constraint will be generated for n >= 0
31
32  end -- class BENCHMARK
```

Figure 33: Schema used to generate benchmark classes.

of the realm update algorithm that one process signals another process the sending of an ACQUIRED or SURRENDER message. This value is an indirect indicator for the degree of universe contention. Note that the implementation is optimized to not signal other processes for every single message sent in order to avoid unnecessary process context switches, so it is not an indicator for the number of messages being sent between processes.

The relative variation of *Overhead* is generally less than a few percent between independent runs. The absolute error for *SignalRate* can be as much as 0.5‰, which corresponds to one signal per two thousand scheduler invocations.

9.1.4 Configuration Options

Our runtime library can be configured by a number of parameters at compile time, some of which require a certain degree of processor support. The *PORTABLE* option selects an implementation that has a high degree of portability across systems that support the POSIX API [IEE96]. The *FASTLOCKS* option supports customized mutexes that were optimized for our purposes instead of using the POSIX implementation. Our optimization essentially removes overhead due to API differences and unnecessary options provided by the POSIX interface and the Linux implementation, but is otherwise a typical mutex implementation. Additionally, it is reduced to a single compare-and-swap call in the absence of contention. The *SMP_SUPPORT* option can be chosen in conjunction with the *FASTLOCKS* option. Disabling it removes an inherent overhead in synchronization mechanisms on symmetric multi-processors using the Intel architecture. Finally, *USER_THREADS* selects an experi-

mental implementation of user level threads[3], which requires compiler support to be preemptive, but eliminates synchronization overhead completely.

9.2 Benchmark Results

To measure the performance of our algorithm as a function of the input parameters, we have constructed a number of scenarios. Each scenario keeps all but one or two of the input parameters fixed, and varies the other parameters along a (usually linear) scale. For each scenario and set of input parameters, we list the output values. We do not explicitly list the value of *SignalRate* in scenarios where no contention occurs, because in that case it is always zero.

Our benchmarks were performed on an Intel Xeon processor, running at an effective frequency of 2.175 GHz, using a Linux operating system (kernel version 2.2.17 configured for uniprocessor use) and the POSIX threads implementation that is part of the GNU libc 2.3.1 implementation. We used the *FASTLOCKS* option and disabled *SMP_SUPPORT*[4], unless specified otherwise.

The first scenario (Table 1 a) attempts to capture the effect of the *DeltaRealm* parameter, all else being equal. For this purpose, we set *Overhead* to zero so as to avoid contention. The *NumberProcesses* parameter was kept at 2 throughout the scenario. Observe that the

[3]The *USER_THREADS* option is based on the portable GNU threads library [Eng00]. It is experimental only insofar as the thread library cannot yet be used in conjunction with the Boehm-Weiser conservative garbage collector [BW88] that we are using, but is otherwise functional.

[4]Enabling *SMP_SUPPORT* on 80x86 processors introduces a fixed overhead per synchronization primitive, caused by prefixing atomic instructions such as compare-and-swap with an instruction that locks the system bus and keeps processor cache lines synchronized. Unfortunately, that overhead varies greatly between processor types, from ~ 30 to ~ 160 clock cycles according to our measurements, and decreases the reproducibility of results.

DeltaRealm	*Overhead*
0	216
1	428
2	564
3	756
4	847
5	1049
6	1290
7	1341

(a)

Overlap	*Overhead*	*SignalRate*
0	1310	0.00‰
1	1376	1.49‰
2	1331	1.34‰
3	1333	1.90‰
4	1356	1.38‰
5	1339	1.77‰
6	1265	1.86‰
7	1330	1.63‰

(b)

Table 1: Scenarios 1 and 2 – No contention and with contention.

value of *Overhead* can be approximated well by a linear function.

The second scenario (Table 1 b) examines various degrees of contention between resources. While still keeping the number of processes at 2, we vary the *Overlap* parameter from 0 through 7. *DeltaRealm* is kept at a constant value of 7, so that we examine every situation from no contention to 100% contention. While the non-zero *SignalRate* indicates that contention does indeed occur, there is no effect on the value of *Overhead* that exceeds the error of measurement.

Finally, the third scenario (Table 2) measures what happens if multiple processes contend for the same set of resources. In this scenario, we track the overhead for 1 through 5 processes contending for the same set of resources. We vary *NumberProcesses* from 1 to 5, and for each value of *NumberProcesses*, we vary *DeltaRealm* from 1 to 5 and from 10 to 30. For each run, *Overlap* will be the same as *DeltaRealm* to create complete contention. This time, we observe a minor performance penalty that increases with the number of processes.

Overhead

NumberProcesses	*DeltaRealm = Overlap*							
	1	2	3	4	5	10	20	30
1	412	599	721	877	1006	1637	3036	4398
2	450	574	760	859	1065	1801	3222	4494
3	466	597	779	880	1073	1741	3432	4585
4	496	613	796	898	1105	1858	3391	4618
5	512	672	840	941	1158	1906	3404	4754

SignalRate

NumberProcesses	*DeltaRealm = Overlap*							
	1	2	3	4	5	10	20	30
1	0.00‰	0.00‰	0.00‰	0.00‰	0.00‰	0.00‰	0.00‰	0.00‰
2	1.14‰	1.19‰	1.51‰	0.98‰	1.11‰	1.65‰	1.28‰	2.50‰
3	3.21‰	3.38‰	3.07‰	4.98‰	3.71‰	5.15‰	7.21‰	6.45‰
4	6.97‰	6.35‰	6.29‰	6.14‰	8.14‰	8.59‰	9.86‰	12.98‰
5	9.61‰	10.98‰	9.17‰	9.72‰	11.69‰	12.29‰	14.10‰	20.58‰

Table 2: Scenario 3 – Process contention.

9.3 Analysis

We have provided performance data for our scheduler on a uniprocessor system using stock hardware and an off-the-shelf operating system, when performing under a variety of circumstances. We now discuss the meaning of these results, what they reflect of our implementation, and why we didn't provide similar data for an SMP architecture.

9.3.1 Uniprocessor Results

In the absence of contention, the realm update algorithm has a cost that is approximately linear in the number of universes involved in an update, with the cost increasing by ~ 160 clock cycles per universe. This is the expected outcome, since the realm update algorithm has a cost that is linear in the number of universes for that case, as long as the underlying graph is tree-shaped.

For two processes and up to 30 shared universes we do not see any increase in cost that significantly exceeds any error due to caching, pipelining, and other sources of error immanent to a modern microprocessor. In particular, for *DeltaRealm* $= 30$ the value of *SignalRate* does not exceed a value of 2.5 per thousand invocations of the realm update algorithm for two threads, compared to a value of 1.14 for *DeltaRealm* $= 1$. That means that while there is minimal overhead due to contention, the cost of contention per universe is negligible in comparison.

For more than two processes, the situation is more complex. For any given number of processes, the increase in *SignalRate* is roughly linear in terms of *DeltaRealm*, and there appears to be a matching increase in *Overhead*, though that is too small compared to the immanent error due to caching, pipelining, etc. to make any reliable statements. The cost per universe appears to be largely dependent on the number of threads involved, and increases non-linearly. However, for all our test cases, the overhead per each additional thread never exceeds 5% of the cost for the case where there is no contention.

To further analyze the case where two or more threads contend for one or more universes, note that the additional cost for contention is dominated by the cost of process context switches. In our implementation, the cost of a context switch is 10–100 times that of the various synchronization primitives that we employ (usually around ~ 4000 clock cycles for a context switch, $\sim 40+$ clock cycles for a synchronization primitive). Thus, even relatively infrequent context switches can contribute significantly to the overall overhead. Context switches can occur either voluntarily by one process transferring control to another, or involuntarily, when a time slice of the OS scheduler has ended.

Involuntary context switches are rare on a uniprocessor, and in our implementation,

voluntary context switches on a uniprocessor can only occur as a side effect of involuntary context switches, due to `ACQUIRE` or `SURRENDER` messages being sent. Furthermore, note that if one process initiates a process switch by signaling the sending of such a message to another process, this can also result in a third process being scheduled instead, causing further contention, as there are now three or more processes negotiating for the acquisition of universes. It is therefore to be expected that the overhead in contention scenarios depends both on the number of universes under contention and the number of processes contending for them.

We conclude that the realm update algorithm has a reasonable cost and scales well on uniprocessors, even for artificially large degrees of contention between processes.

9.3.2 SMP Architectures

We have not provided data for our benchmarks when running on an SMP architecture, because they expose a pathological case of SMP scheduling. For example, when running two processes with $\mathit{DeltaRealm} = \mathit{Overlap} = 1$ and $\mathit{AvgIntervalLength} \approx 1000$ on the same architecture with an SMP-capable OS kernel, we have $\mathit{Overhead} > 5000$ clock cycles, over ten times times the uniprocessor cost, and $\mathit{SignalRate} > 950$ per thousand realm updates, meaning a process context switch for almost every invocation. Of the signals delivered, over 99.9% were for universe acquisitions, and less than 0.1% for *SURRENDER* messages. Therefore, the increased overhead is almost entirely attributable to the increase in *SignalRate* and the concomitant context switches. In particular, the context switch overhead alone exceeds the synchronization cost on uniprocessors even for very large realm sizes.

This massive increase in overhead is not a shortcoming of the realm update algorithm, but simply due to lock contention; a similar increase in overhead could be observed when replacing the realm update procedure with code that locks a semaphore when `c = true` and unlocks it when `c = false`. To understand the reason, observe that on a uniprocessor, once a process P relinquishes a resource and transfers control to another process Q, P will be suspended for a full time slice, where it can't interfere with Q's operation. On an SMP architecture under the same circumstances, P continues running, until it is blocked when attempting to acquire the shared resource again. Essentially, on an SMP architecture under high contention, almost each resource acquisition carries the additional cost of a context switch.

Observe that this problem need not occur if resource management is not completely fair. A uniprocessor scheduler does not have the problem because Q is allowed to acquire the resource several times before P can attempt it again. This approach sacrifices fairness for performance. To obtain similar performance in the SMP case under high contention, a resource management algorithm will have to give up a degree of fairness to avoid the above scenario.

This phenomenon occurs in practice as well. Previous research has found that applications that use blocking synchronizations can spend a third of their time on context switching [KLMO91], while another study [JS99] that used the SPLASH-2 benchmarks found that for some applications half the time was spent waiting on synchronization. A commonly proposed solution [KLMO91, TF02] is to use busy waiting ("spinning") instead of idle waiting to eliminate context switches in those cases where context switch overhead dominates execution time. To study this, we implemented an experimental spinning approach for

our realm update algorithm by inserting a busy waiting loop that delayed until `u.owner` $\neq$ `nil` at the beginning of `request`. By doing this, the overhead of the realm update became comparable to the uniprocessor case (within 10%). Unfortunately, this simple solution can potentially lead to starvation, does not scale well for multiple universes, and can unnecessarily occupy processors on multi-user machines. An alternative solution to the problem of excessive context switch overhead is to use user-level threads (which can be enabled in our implementation by setting the `USER_THREADS` option in the runtime system), which are limited to a single processor, but consequently cannot exploit the parallelism of SMP systems. Further research is needed to determine the best solution to deal with such scenarios.

Chapter 10

Conclusion and Future Research

We have presented an approach to synchronization that relies on the specification of declarative contracts, rather than the embedding of synchronization mechanisms in the procedural code of a class. This approach can be integrated easily in object-oriented languages. We have provided algorithms for the most difficult parts of its implementation, validated their efficiency, and provided a set of language extensions for Eiffel that allow a programmer to use the model. The remaining open questions are the relative power of the universe model relative to existing approaches and what additional features may have to be supported by a contractual model in the future.

10.1 Comparison of the Universe Model to Existing Approaches

The universe model does not provide capabilities for every single concept that has been discussed so far, especially not all of the features of the synchronization mechanisms and

contractual models presented in Section 2.6.2. Sather, CEE, and SCOOP have features for the explicit composition of contracts that are not covered by the universe model, and which we discuss first. Second, we consider the issue of intra-object synchronization contracts, which are provided by the synchronization ring model, but not by the universe model. Conversely, the universe model supports the implicit composition of synchronization contracts, which is not available in any of the other approaches.

10.1.1 Explicit Composition beyond the Universe Model

A distinguishing feature of Sather that cannot currently be replicated in the universe model is the else clause of its multi-branch lock statement. It specifies an alternate execution path that can be taken when none of the available branches can be taken. The universe model as a contractual model does not support such semantics. [FP97] describe three possible alternative semantics for the else-branch, none of which is clearly superior to all of the others. Should semantics similar to the else-branch be found to be desirable, a contractual approach would be to use *timeout contracts*, which are contracts that are satisfied after a certain time, but not before. For instance, the contract

$$C \rightarrow S_1 \vee C \rightarrow S_2 \vee \mathit{timeout}(t)$$

can be satisfied by acquiring either S_1 or S_2 within t time units, or after the expiration of $t \geq 0$ time units if neither S_1 nor S_2 can be acquired before the timeout expires.

Beyond the synchronization contracts allowed by the universe model, both SCOOP and CEE allow the programmer to express the obligations of a conditional synchroniza-

tion contract that involves a predicate that can depend on the state of several shared objects. Conversely, the universe model only allows for synchronization contracts of the form $C \overset{P(C,S)}{\rightarrow} S$, where $P(C,S)$ depends only on the state of C and S. Restricting conditional contracts in such a fashion was a conscious design decision. It simplifies the correct implementation of when clauses considerably, which is important, given the already considerable complexity of the basic deadlock avoidance/recovery algorithm. There is no principal problem in extending the language of concurrency constraints to allow the condition of when clauses to depend on the state of additional universes, except that implementation of such a scheme would be non-trivial. We believe that extending the language of concurrency constraints to cover more powerful conditional contracts should not be done until there is sufficient evidence that the additional power is useful in practice.

10.1.2 Intra-object Contracts

A type of synchronization contract that we have not studied in this thesis is that of *intra-object synchronization contracts*, which are supported by the synchronization ring model [Hol99]. Whereas our existing classification of synchronization contracts assumes that a process has either exclusive access to a supplier or no access at all, intra-object synchronization contracts specify if (and under what circumstances) multiple clients that are owned by different processes can access the same shared supplier concurrently (a classic example of this type of contract are path expressions [CH74]). A common case of intra-object concurrency is that of the *multiple readers*, where multiple processes access a shared supplier, as long as they only read the supplier's data, but do not modify it. This is obviously

thread-safe and can significantly improve the level of concurrency.

One way to specify intra-object synchronization contracts is the concept of *ports* [Hol99]. Using this notion, shared objects are accessed *via a specific port*. For example, there might be both a read-port R, which allows only read-only access, and a write-port W, which allows read-write access. More than one process can access the same supplier via port R, but if one process accesses the supplier via port W, then no other process can access the same supplier.

While intra-object synchronization is well understood for the multiple readers problem, one can specify more complex requirements for intra-object concurrency, which significantly complicates the negotiation of synchronization contracts. In particular, a general approach would allow the specification of an arbitrary set of ports $P_1, \ldots, P_k$. One has to then define which ports can be accessed concurrently and which cannot, and has to ensure that no unwanted interference can occur, which is a non-trivial problem. As [Hol99] notes, however, intra-object concurrency is required only infrequently in practice. Considering also that intra-object contracts require a more complex runtime system to negotiate them, possibly to the point of affecting its efficiency, we have deferred the specification and implementation of intra-object synchronization contracts to future work.

10.1.3 Implicit Composition of Contracts in the Universe Model

In addition to providing synchronization contracts as a first-class language feature, the universe model exceeds the contractual capabilities of the mechanisms discussed so far by allowing for the implicit composition of contracts. Whereas previously discussed ap-

proaches assume that the client of a synchronization contract is a process, job, or transaction, the universe model assumes that the client of a synchronization contract is a shared resource (specifically, a universe) owned by a process. By this token, the supplier of one synchronization contract can become the client of a second (and possibly third, fourth, etc.) synchronization contract. This *chaining* of synchronization contracts, as described by the notion of *accessible* links in Chapter 4 is not possible if the client and the supplier of a synchronization contract are a process and a shared resource, respectively.

In addition to being able to construct chains of synchronization contracts, one has also to be able to evaluate the composition of synchronization contracts that occur along these chains, which is only feasible in a contractual model. For example, in order for the client C of an exclusion contract $C \rightarrow S$ to know about the behavior of S with respect to synchronization, the essential information about that behavior must be available through the interface of S. If S were to employ a procedural synchronization mechanism, C would have to perform procedural analysis of the code of S in order to infer the same information, which is both undecidable in the general case and violates basic information hiding principles. Thus, we conclude that language support for implicit composition is only possible if a supplier S informs its clients about the synchronization requirements S has when it is itself acting as a client. In other words, S must publish its contractual relationships.

10.2 Open Research Questions

We conclude this discussion by describing two open research problems. First, our classification scheme provides only one way to represent disjunctions, namely the $\vee$ operator. As

noted already in Section 10.1.1, this is insufficient to represent the else-branch of Sather's lock statement. But more generally, there are several feasible interpretations of the semantics of the disjunction of synchronization contracts, which creates an inherent ambiguity. The second problem is that the negotiation of certain composite conditional contracts may require influencing the OS scheduler in non-trivial ways. Unlike the first, the second one is a concern for procedural and contractual approaches to synchronization alike, and may simply prove to be infeasible. We now address these two problems in turn, and show for the first one how it can be solved by extending the classification scheme for synchronization contracts.

10.2.1 Treatment of Disjunctions

The ambiguity of composing synchronization contracts by disjunction arises from a subtle fairness problem in their interpretation. As an example, consider the composite contract $C \rightarrow S_1 \vee C \rightarrow S_2$. Each time that this contract is negotiated, the system has to acquire either S_1 or S_2. One has to avoid always choosing the same supplier S_i to avoid starving the other supplier. Were one to acquire the first supplier (either S_1 or S_2) that became available, then it would be possible that S_1 would always be selected and S_2 never, or vice versa. On the other hand, acquiring the first supplier to become available will likely result in better performance. Judging from just the synchronization contract without additional information, either interpretation may be the one intended by the programmer.

Both interpretations can feasibly be implemented in practice as follows and are therefore not purely theoretical considerations. If one has a deadlock- and starvation-free algo-

rithm to implement synchronization contracts composed by conjunction, it is always possible to derive an algorithm for composition by disjunction as well and either interpretation of the $\vee$ operator above has an effective implementation. To illustrate, consider a contract $A_1 \vee \ldots \vee A_m$, where the A_i are conjunctions of arbitrary basic contracts. For a *completely fair* algorithm that does not exploit parallelism, select one of the A_i at random, and attempt to satisfy it. For a maximally concurrent (but possibly unfair) algorithm, attempt to satisfy the conjunction $A_1 \wedge \ldots \wedge A_m$. As soon as there is a j such that A_j is satisfied, release all resources that are not needed by A_j. We call this the *eager* approach to negotiating disjunctions. Since neither approach can be disregarded as infeasible, the question remains which one was intended by the programmer.

In order to disambiguate between these (and possibly other) interpretations, not all types of disjunctions can be represented by the same operator $\vee$. Instead, a plurality of operators may be needed to encode the intent of the software designer. A related notion exists in the concurrent specification language CSP [Hoa85], which has two distinct operators to describe non-deterministic choice. Likewise, providing an *or-else* operator would allow synchronization contracts to obviate Sather's lock statement entirely. Whether such an extension of the language of synchronization contracts is necessary, or one interpretation can satisfy most situations that occur in practice, can only be answered by experience.

Regarding existing practice of dealing with the ambiguity of disjunction, those synchronization mechanisms that effectively implement composition by disjunction seem to only provide one interpretation of disjunction, which is either the eager option [BBF01], leaving fairness management up to the programmer, or a hybrid approach. An example of a hybrid approach is the implementation used by the programming language Sather [FP97]. Sather's

algorithm attempts to satisfy each of the disjuncts (in random order) in turn. After attempting to satisfy one of them, the algorithm waits for a specified time interval (which increases with each attempt) before trying the next alternative. While this does not absolutely guarantee fairness, it makes starvation unlikely, as long as locks are not held for a time exceeding the initial time interval. The algorithm that we described in this thesis uses the completely fair approach, though chiefly to simplify its presentation. An implementation like Sather's would be a straightforward extension of the existing implementation.[1]

10.2.2 Composition of Conditional Contracts

The fair negotiation of composite conditional contracts creates a non-trivial scheduling problem. Consider the composite contract

$$C \xrightarrow{P} S_1 \wedge C \xrightarrow{Q} S_2.$$

Let $P(t)$ and $Q(t)$ be the value of P and Q, respectively, at time t, and suppose that $P(t)$ and $Q(t)$ both become true infinitely often, and independently of one another. However, whether $P(t) \wedge Q(t)$ ever becomes true depends entirely on scheduling. It is possible for scheduling to occur in such a fashion that $P(t)$ is true only when $Q(t)$ is false, and vice versa. Thus, any negotiation mechanism that tries to satisfy the composite contract has to do one of two things. It can either simply *observe* $P \wedge Q$ without interfering with the scheduling, and wait for $P \wedge Q$ to become true, or it can attempt to *control* the execution of processes that alter P and Q, steering them to a point where $P \wedge Q$ becomes true. The observing approach

[1] One simply has to reevaluate `accessibles` after each invocation of `dfsRestart` in Chapter 6.

has the potential to lead to starvation; the controlling approach is difficult to implement in the general case. Therefore, most existing approaches that allow for composite contracts either use a strictly observing approach (such as [CW02]) or use an observing approach augmented by some simple timing heuristics that can delay processes to apply a minimum of control (such as [FP97]). Our implementation likewise provides means for an observational approach with timing-based heuristics; we consider a fully controlling approach for the conjunction of conditional contracts to be an open research question, which may prove to be infeasible.

10.3 Outlook

Our long-term research vision is that the use of synchronization contracts may fundamentally simplify the development and improve the reliability of multi-threaded shared-memory systems. Specifically, we believe that declaratively specified (and automatically negotiated) contracts will obviate the need for complex ad hoc synchronization mechanisms and protocols, thereby reducing the dominant source of complexity in these programs. As the use of powerful synchronization contracts is a relatively new programming paradigm, validating the long-term vision will require more research into building systems using this paradigm. Thus, we cannot yet make any general claims on this point.

APPENDICES

Appendix A

Low-Level Algorithmic Details

A.1 The Message Passing Subsystem

A.1.1 Checking for realm completion

Figure 34 depicts the function `realmComplete`, which performs two services for the `completeRealm` algorithm. Procedure `completeRealm` invokes `realmComplete` to decide whether to terminate the loop over pending requests for universe acquisition. When this loop terminates, the realm must be officially completed by moving all of the acquired universes into the core of `p`. For efficiency reasons, `realmComplete` actually performs the movement into the core in addition to returning the boolean value true to signal termination of the acquisition loop in `completeRealm`

More precisely, `realmComplete` (p) returns true if and only if:

1. $\forall u : \langle p, u \rangle \in$ Negotiating $\bullet \langle p, u \rangle =$ Owned and

2. and p has no knowledge of an imminent transition within any of its negotiations.

```
proc signal( p : Process,
             msg : MessageType,
             u : Universe )
begin
    enqueueMsg(msgQ(p), ⟨ msg, u ⟩);
end

function realmComplete( p : Process)
returns boolean is
    var pendingMessages: boolean;
begin
    if (pending(p) > 0) then
        return false;
    end;
    LOCK(p);
    pendingMessages := true;
    if (empty(msgQ(p))) then
        pendingMessages := false;
        start_time(p) := current_time();
    end;
    UNLOCK(p);
    return ¬ pendingMessages;
end

function getMessage( p : Process )
returns MessageType × Universe is
    var msg: MessageType; u: Universe;
begin
    wait until (¬ empty(msgQ(p))) then
        LOCK(p);
        msg, u := dequeueMsg(msgQ(p));
        UNLOCK(p);
    end;
    return ⟨ msg, u ⟩;
end

function inMigrating( u : Universe,
                      p : Process )
returns bool is
begin
    return migrating(u) ∧ owner(u) = p;
end

function inOwned( u : Universe,
                  p : Process )
returns bool is
begin
    return ¬migrating(u) ∧ owner(u) = p;
end
```

Figure 34: Various algorithms for message passing and negotiation-state observation.

This condition is checked by evaluating whether p has any pending acquisition requests or pending messages. If there are no pending acquisition requests or messages in the queue, then all of the requested universes are moved into the core by updating the start time of p.

A.1.2 Message Transmission

Inter-process messages are transmitted by procedure `signal` and retrieved by procedure `getMessage`. These procedures implement message passing by appending or removing messages from a process' message queue. Procedure `signal` is trivial. Procedure `getMessage` retrieves the first message on the queue, blocking if the queue is empty. By our definition of the **wait until** primitive, p is locked for the duration of the test.

A.1.3 In-state predicates

Functions `inMigrating(u, p)` and `inOwned(u, p)` return true if $\langle p, u \rangle$ = Migrating and $\langle p, u \rangle$ = Owned respectively. The actual conditions checked are explained in Section 5.2.1.

A.2 Releasing and Requesting Universes

A.2.1 Manipulating the Set of Blocked Universes

Procedures `recordBlocked` and `recordUnblocked` add (respectively remove) a universes to (respectively from) the set of blocked universes of a process (CalloutFigure-fig.request.release). Both implementations are straightforward. Observe that if the set is

```
proc request( u : Universe,
              p : Process,
              rc : Natural ) is
var oldHead : Process := head(reqQ(u));
begin
   if (contains(reqQ(u), p)) then
      updateQueueRC(reqQ(u), p, rc);
   else
      enqueue(reqQ(u),
         start_time(p), p, rc);
      pending(p) := pending(p) + 1;
      if (oldHead ≠ ⊥ ∧
         head(reqQ(u)) ≠ oldHead) then
         recordBlocked(u, oldHead);
      end;
   end
end

proc recordBlocked( u: Universe,
                    p : Process ) is
begin
   LOCK(p);
   blocked(p) := blocked(p) + { u };
   UNLOCK(p);
end
```

```
proc releaseFromQueue( u : Universe,
                       p : Process,
                       rc : Natural ) is
var preempting : bool;
begin
   LOCK(p);
   preempting := u ∉ blocked(p);
   UNLOCK(p);
   updateQueueRC(reqQ(u), p, -rc);
   if (¬contains(reqQ(u), p)) then
      pending(p) := pending(p) - 1;
      recordUnblocked(u, p);
      if (preempting
         ∧ ¬ empty(reqQ(u))
         ∧ start_time(head(reqQ(u)))
              < acquisition_time(u))
      then
         recordUnblocked(u,
              head(reqQ(u)));
      end
   end;
end

proc recordUnblocked( u: Universe,
                      p : Process ) is
begin
   LOCK(p);
   blocked(p) := blocked(p) - { u };
   UNLOCK(p);
end
```

Figure 35: Support procedures for requesting and releasing universes.

changed for a process that is not the currently executing process, then it might trigger the **wait until** test in function `getMessageCooperatively` (Figure 20). Thus, depending on the implementation of **wait until**, this condition might need to be signaled explicitly.

A.2.2 Requesting a Universe

Procedure `request` enqueues a request, by a process `p`, for a universe `u` with reference count `rc` (Figure 35). The procedure begins by storing the current head of the queue in a local variable (`oldHead`) for later reference. It then tests if the queue already contains the process `p`, and if so, the reference count in the queue for that process is updated, and no further action is taken. Otherwise, `p` is inserted into the queue according to its `start_time`, and `pending(p)` is incremented to reflect that there is one more outstanding request.

Inserting a new process into the queue could make the new process the head of the queue, if for example the priority of `p` exceeds that of `oldHead`. In this situation, `oldHead` must be informed that its negotiation for `u` is now blocking. Observe that it is possible that the negotiation was already blocking prior to the insertion, but the negotiation could also have been preempting.

A.2.3 Releasing a Process from a Queue

Procedure `releaseFromQueue` removes from the request queue of *u* some number (*rc*) of link requests on behalf of process *p*. This removal is accomplished by decrementing the reference-count entry in the tuple associated with *p*. If this operation removes all pending link requests on behalf of *p*, then the tuple associated with *p* is removed, in which case

the number of pending requests for p must be decremented by one and u must be removed from `blocked(`p`)`. The operation becomes more complicated if the entry for p was at the head of the queue and if $\langle p, u \rangle$ was preempting because then removing the entry for p might affect the blocked status of a another process in the queue.[1] Specifically, if what was the second entry in the queue becomes the new head of the queue, the process associated with this second entry might need unblocked. To test this case, the procedure sets the flag `preempting` to record whether $\langle p, u \rangle$ = **Preempting**. This implies that p was at the head of the queue prior to the decrementing of link requests. If p was preempting, the queue is not empty, and the start time of process q that is the new head of the queue is less than the acquisition time of the universe, then $\langle q, u \rangle$ must transition from **Blocked** to **Preempting**.

[1] Notice that accessing the start time of the process at the head of the queue must be synchronized. Whereas the queue is ordered by the start time of the process component of its elements, we assume here that we can access this time from the tuple at the head of the queue, as opposed to accessing it from the process component of this tuple.

Bibliography

[ABAK95] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy. Managing concurrent activities in collaborative environments. In *Conference on Cooperative Information Systems*, pages 112–124, 1995.

[ACM87] R. Agrawal, M. J. Carey, and L. W. McVoy. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Transactions on Software Engineering*, SE-13(12):1348–1363, December 1987.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–248, July 1997.

[BA90] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[BAL+01] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 92–103. ACM Press, 2001.

[BAR+03] D. F. Bacon, C. R. Attanasio, V.T. Rajan, S. E. Smith, and H. B. Lee. A pure reference counting garbage collector. Submitted for publication, available at http://www.research.ibm.com/people/d/dfb/papers.html, 2003.

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[BB98] V. Barbosa and M. Benevides. A graph-theoretic characterization of and-or deadlocks, 1998.

[BBD82] R. Balter, P. Berard, and P. Decitre. Why control of the concurrency level in distributed systems is more fundamental than deadlock management. In *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 183–193. ACM Press, 1982.

[BBF01] V. C. Barbosa, M. R. F. Benevides, and A. L. Oliveira Filho. A priority dynamics for generalized drinking philosophers. *Information Processing Letters*, 79(4):189–195, 2001.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, 1987.

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

[BJR99] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, Reading, Massachusetts, 1999.

[BK85] J. A. Bergstra and J. W. Klop. Algebra for communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[BK91] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[BKH97] A. Burger, V. Kumar, and M. L. Hines. Performance of multiversion and distributed two-phase locking concurrency control mechanisms in distributed databases. *Information Sciences*, 96(1+2):129–152, 1997.

[Blo79] T. Bloom. Evaluating synchronisation mechanisms. In *Seventh International Symposium on Operating System Principles*, pages 24–32, 1979.

[Bri72] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BS00] R. Behrends and R. E. K. Stirewalt. The universe model: An approach for improving the modularity and reliability of concurrent programs. In D. S. Rosenblum, editor, *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, volume 25, 6 of

ACM Software Engineering Notes, pages 20–29. ACM Press, November 8–10 2000.

[BW88] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18:807–820, 1988.

[Car90] D. Caromel. Concurrency: An object-oriented approach. In Jean Bezivin, Bertrand Meyer, and Jean-Marc Nerson, editors, *TOOLS 2 (Technology of Object-Oriented Languages and Systems)*, pages 183–197. Angkor, 1990.

[CG92] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[CH74] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Operating Systems, Proceedings of an International Symposium*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer-Verlag, 1974.

[Che95] I.-R. Chen. Stochastic Petri net analysis of deadlock detection algorithms in transaction database systems with dynamic locking. *The Computer Journal*, 38(9):717–733, 1995.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *CACM*, 24:198–206, 1981.

[CM84] K. M. Chandy and J. Misra. The drinking philosopher's problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[CPF99] C. D. Carothers, K. S. Perumalla, and R. Fujimoto. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *Winter Simulation Conference*, pages 1624–1633, 1999.

[CS99] M. Choy and A. Singh. Dynamic resource allocation using views. Technical Report TRCS99-36, Department of Computer Science, University of California, Santa Barbara, 1999.

[CT90] C. Critchlow and K. Taylor. The inhibition spectrum and the achievement of causal consistency. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, pages 31–42, 1990.

[CW02] M. Compton and R. Walker. A run-time system for SCOOP. *Journal of Object Technology*, 1(3):119–157, 2002. Special issue: TOOLS USA 2002 proceedings.

[Dij68] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press N.Y., 1968.

[DYK01] L. DeMichiel, L. Ü. Yalcinalp, and S. Krishnan. The Enterprise JavaBeans 2.0 specification, 2001. http://java.sun.com/products/ejb/docs.html.

[EAWJ96] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[Eng00] R. S. Engelschall. Portable multithreading — the signal stack trick for user-space thread creation. In USENIX, editor, *2000 USENIX Annual Technical Conference: San Diego, CA, USA, June 18–23, 2000*, pages 239–249, Berkeley, CA, USA, 2000. USENIX.

[FA93] S. Frølund and G. Agha. A language framework for multi-object coordination. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, pages 346–360. Springer-Verlag, 1993.

[FHRT92] P. A. Franaszek, J. R. Haritsa, J. T. Robinson, and A. Thomasian. Distributed concurrency control with limited wait-depth. In *12th International Conference on Distributed Computing Systems*, pages 160–169, Washington, D.C., USA, June 1992. IEEE Computer Society Press.

[Fid88] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1), February 1988.

[FK01] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. of the ESEC and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2001.

[Fos96] I. Foster. Compositional parallel programming languages. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, July 1996.

[FP97] C. Fleiner and M. Philippsen. Fair multi-branch locking of several locks. In *International Conference on Parallel and Distributed Computing and Systems*, pages 537–545. IASTED/ACTA Press, 1997.

[FR85] P. Franaszek and J. T. Robinson. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems (TODS)*, 10(1):1–28, 1985.

[GCCC85] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 255–263, University Park, Pennsylvania, August 1985. IEEE.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.

[GJSB96] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 1996.

[Han99] P. Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34, 1999.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice/Hall International, Englewood Cliffs, New Jersey, 1985.

[Hol99] D. Holmes. *Synchronisation Rings - Composable Synchronisation for Object-Oriented Systems*. PhD thesis, Macquarie University, Sydney, October 1999.

[HZ92] M. Hsu and B. Zhang. Performance evaluation of cautious waiting. *ACM Transactions on Database Systems*, 17(3):477–512, September 1992.

[IEE96] IEEE. *1996 (ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language].* IEEE, New York, NY, USA, 1996.

[INM88] INMOS, Ltd. *OCCAM 2, Reference Manual.* Prentice-Hall, 1988.

[Jef85] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[JL96] R. Jones and R. Lins. *Garbage Collection.* John Wiley and Sons, New York, 1996.

[JP93] G. Jalloul and J. Potter. A separate proposal for Eiffel. In *Proceedings of the 2nd Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific '93).* Prentice Hall, 1993.

[JS99] D. Jiang and J. P. Singh. Scaling application performance on a cache-coherent multiprocessor. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 305–316. ACM Press, 1999.

[Kai95] G. E. Kaiser. Cooperative transactions for multiuser environments. In *Modern Database Systems*, pages 409–433. ACM Press and Addison-Wesley, 1995.

[KLMO91] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the13th ACM Symposium on Operating Systems Principle*, pages 41–55. ACM Press, 1991.

[LC01] B. Leuf and W. Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet.* Addison-Wesley, 2001.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In Michael Cosnard, Yves Robert, Patrice Quinton, and Michael Raynal, editors, *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V. (North Holland), Amsterdam, 1989.

[Mat93] F. Mattern. *Distributed Control Algorithms (Selected Topics)*, pages 167–185. Springer-Verlag, 1993.

[Mey92] B. Meyer. *Eiffel: the Language.* Prentice Hall, 1992.

[Mey93] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Number 94 in LNCS. Springer-Verlag, Berlin, 1980.

[Mit95] S. Mitchell. *TAO — A Model for the Integration of Concurrency and Synchronisation in Object-Oriented Programming*. PhD thesis, University of York, UK, 1995.

[MWL90] A. Martinez, R. Wachenchauzer, and R. D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34(1):31–35, 1990.

[Nie95] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[NM92] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[Omo91] S. M. Omohundro. The Sather language. Technical report, International Computer Science Institute, 1947 Center Street, Suite 600, Berkely, California 94704, 1991.

[PA98] G. A. Papadopoulos and F. Arbab. Coordination models and languages. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 46, pages 329–400. Academic Press, 1998.

[Pat90] P. C. Patton. *The Strand88 Concurrent Programming Environment*. Superperformance Computing Service/Strand, London, United Kingdom, March 1990. Brief No. 39.

[Pur94] J. M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[Rhe98] I. Rhee. A modular algorithm for resource allocation. *Distributed Computing*, 11(3):157–168, 1998.

[RSL78] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, June 1978.

[SBD03] R. E. K. Stirewalt, R. Behrends, and L. K. Dillon. A model-based semantics for synchronization contracts in object-oriented languages. Submitted for publication in the Elsevier journal *Science of Computer Programming*. An online copy may be found at http://www.cse.msu.edu/~stire/Papers/scp03.pdf, 2003.

[SKT+96] J. Siegel, A. Klein, A. Thomas, D. Frantz, and H. Mirsky. *CORBA Fundamentals and Programming*. John Wiley and Sons, Inc, New York, 1996.

[SM94] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[SPG91] A. Silberschatz, J. L. Peterson, and P. Galvin. *Operating Systems Concepts*. Addison-Wesley, third edition edition, 1991.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliff, NJ, 1992.

[TF02] D. Tsafrir and D. G. Feitelson. Barrier synchronization on a loaded SMP using two-phase waiting algorithms. In *International Parallel and Distributed Proceedings Symposium*. IEEE, April 2002.

[TGC93] E. Tryggeseth, B. Gulla, and R. Conradi. Software configuration management in PROTEUS. In *Proceedings of the 4th International Workshop on Software Configuration Management (Preprint)*, pages 232–240, Baltimore, Maryland, 1993.

[Wil92] P. R. Wilson. Uniprocessor garbage collection techniques. In Y. Bekkers and J. Cohen, editors, *Proc. of the International Workshop on Memory Management*, Lecture Notes in Computer Science 637, pages 1–42. Springer–Verlag, 1992.

[YT87] Y. Yokote and M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, Cambridge, Mass., 1987.