A model-based semantics for synchronization contracts in object-oriented languages

R. E. K. Stirewalt, Reimer Behrends, Laura K. Dillon
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48864
{stire, behrends, ldillon}@cse.msu.edu

Abstract

We previously developed a powerful model of synchronization contracts in strongly typed object-oriented languages. The model separates the specification of concurrency and synchronization properties from the specification of a program’s core functionality, with the former expressed in a declarative constraint language and the latter expressed in a traditional object-oriented programming language. One danger in this approach is the potential for subtle semantic interactions between the base-language program and the run-time machinery that is introduced to implement the declarative specifications. To address this problem, we formalize the semantics of our model in a manner that is independent of the particular base language being extended. Moreover, we structure the formal semantics to separate the base-language and concurrency-management operations into distinct views, which compose by conjunction. This clean separation of concerns addresses the afore-mentioned danger. On another level, the paper demonstrates how to structure the formal semantics of a contract model in isolation in a way that integrates cleanly with most strongly typed object-oriented languages. The semantics of synchronization contracts is formalized in the Z notation.

1 Introduction

Concurrent applications in which multiple threads access objects in shared memory are notoriously difficult to design. Without thread synchronization, concurrent access to shared objects can lead to race conditions [39], and incorrect synchronization logic can lead to starvation and deadlock. Moreover, these difficulties confound the development of reusable software modules because synchronization policies and decisions are difficult to localize into a single software module. Thus, a module can easily implement a synchronization policy that satisfies safety and liveness requirements in some usage contexts but that fails to satisfy the same requirements in other contexts. To address these concerns, researchers have recently looked to extend and articulate Meyer’s design by contract method to address the issues of concurrent thread synchronization [8, 35]. Contracts used toward this end are called synchronization contracts,
and in prior work [4], we developed a powerful model of synchronization contracts that integrates well with object-oriented languages. We have incorporated this model into two languages, Ruby [50] and Eiffel [35], and we believe that similar extensions could apply to most strongly typed object-oriented languages. This paper provides a formal semantics of this contract model. On another level, this paper demonstrates how to structure the formal semantics of a contract model in isolation in a way that integrates cleanly into a wide range of existing languages. The separation of concerns afforded by this technique improves the understandability and reusability of our contract model and may provide similar benefits to other contract models.

1.1 Synchronization contracts

Briefly, the term contract refers to a formal agreement between a software module, called the supplier, that provides a service and other modules, called the clients, that use the service. A familiar example is the use of operation pre-conditions to specify the assumptions made by a supplier and operation post-conditions to indicate the guarantees that the supplier ensures when the operation is invoked under these assumptions. A contract-aware module is designed to assume the rights and ensure the responsibilities specified in its contract. Contract-aware modules tend to be dramatically simpler than their contract unaware equivalents, which leads to a design rule that Meyer calls “guarantee more by checking less” [35]. By explicitly specifying client and supplier rights and responsibilities, contracts enable the design of highly efficient service implementations, facilitate reasoning about the correctness and reliability of module interactions, and permit compile-time optimizations. More recently, Beugnard and others have investigated how contract awareness can be used to dynamically negotiate interactions among clients and suppliers in order to satisfy quality-of-service requirements [8]. Our work builds on these ideas by providing a powerful model of contract-aware modules in which a run-time system dynamically negotiates interactions to guarantee safety requirements, specifically mutual exclusion, and freedom from certain kinds of deadlock and starvation.

A synchronization contract describes client and supplier rights and responsibilities when performing operations that make use of shared resources. For example, in a concurrent client-server system with multiple client and server objects, a synchronization contract might assert that a client can perform a sequence of operations on one or more servers without interference by other clients. Traditional models for expressing synchronization requirements include monitors [23] and path expressions [13], both of which have been integrated into modern languages. Meyer proposed a contract model, called SCOOP [35], in which operation pre-conditions are interpreted as guards, such that an operation invocation will block until its pre-condition is satisfied. Models, such as SCOOP, assign much of the responsibility for dynamic resource negotiation to a run-time system, which schedules processes based on deadlock- and starvation-avoidance heuristics. Our model builds on these ideas and results.

1.2 A model of synchronization contracts

We recently developed a high-level model of synchronization contracts for multi-threaded object-oriented programs [4]. The model separates the specification of synchronization contracts from the specification of a program’s core func-
tionality, with the former expressed in a declarative constraint language and the later expressed in a traditional object-oriented programming language. Contracts in our model are expressed as *concurrency constraints*, which specify the conditions under which *synchronization units* require exclusive access to one another; here, a synchronization unit is just a module containing data that should be accessed together (i.e., as a “unit”). The contract is interpreted as follows: If the developer of a synchronization unit ensures that her code accesses other units only under the conditions specified in the contract, then she may assume exclusive access to these units while those conditions hold. For example, in a producer/consumer system, we would declare separate synchronization units to contain the producer module, the consumer module, and a communication channel module. The contracts would then specify that, when the producer puts data into the channel, it needs exclusive access to the channel and the channel must not be full and, similarly, when the consumer extracts data from the channel, it needs exclusive access to the channel and the channel must not be empty.

In our contract model, synchronization logic is derived from the contract specifications associated with the interfaces of units rather than programmed explicitly in the bodies of modules. Briefly, processes operate in disjoint data spaces, called *realms*, which grow and shrink over the lifetime of a program. A synchronization unit is then just a group of objects that migrate together among realms. The run-time system manages the dynamic migration of units and ensures that the realms of different processes never overlap. In controlling object migration and process scheduling, the run-time system guarantees the invariance of the concurrency constraints in a program, thereby enforcing the contracts. This functionality is achieved without the programmer needing to write code that explicitly manipulates realm, unit, or thread representations. In the producer/consumer system, a producer unit always belongs to the realm of the producer process and a consumer unit always belongs to the realm of the consumer process, while a channel unit migrates between the realms of these processes. For example, if the channel contains data, the consumer is in a state where it is ready to extract data, and the channel is not in the producer’s realm, then the run-time system automatically migrates the channel into the realm of the consumer process. At any time, the channel unit can either be in the realm of the producer process or the consumer process, or be outside of both realms; but it can never be simultaneously in both realms. Should both processes need to access the channel, one of them will block until the channel migrates out of the realm of the other process and the channel is in a state in which the access is allowed.

Contract awareness under this model provides three major benefits to developers. First, a developer may assume that contract-conforming code will have exclusive access to shared resources; she does not need to program the low-level synchronization required to ensure this exclusive access. The code is greatly simplified because the synchronization logic is not interleaved with functional code, but is factored out and expressed declaratively. Second, contract violations can be automatically checked at run-time [6]. If a client violates its contractual obligations by accessing a supplier without negotiating a contract for that access, a run-time exception is raised. Third, synchronization contracts in our model are composable, which means simple contracts can be combined to form more complex contracts. This composition can be done either explicitly by the programmer or implicitly by the run-time system, which tracks interdependencies among contracts. Thus, the synchronization requirements of individual modules of

---

1 for brevity, we often refer to a synchronization unit as just a unit.
large programs are properly localized within those modules and later combined with those of other modules, thereby enabling the safe reuse of contract-aware modules. Efficient distributed algorithms are used to negotiate composite contracts in an atomic manner while avoiding, to the extent possible, deadlock and starvation [6]. These three benefits—simplification of code, protection of the integrity of shared data, and enforcement of complex, modularized synchronization requirements—ease the design of concurrent systems with complex synchronization requirements.

For a more complex example, consider a multi-threaded configurable web server with a pipelined design. Such a server processes an HTTP request in a pipeline consisting of several stages, which include mapping URIs to files or a server-side script, performing authentication, selecting the right content in a multi-lingual environment, using an internal or external scripting engine to create dynamic content, and postprocessing the content to adjust formatting or to compress the result. In addition, users can create their own modules that add stages to this pipeline. Many stages in the pipeline will need access to shared content with differing synchronization requirements. For instance, a URI-rewriting scheme may cache results for use by multiple threads; an authentication mechanism may depend on a non-rentrant third-party C library, which can thus be used by only one thread at a time; the internal scripting engine may require too much memory or too many file descriptors to replicate the library for each thread; and the scripting engine itself may need access to other shared resources, such as the authentication library. Thus, while the pipelined structure should allow for a fair amount of parallelism, execution must be regulated to avoid conflicts between individual stages. Each stage has its own exclusion requirements, which typically encompass one or more shared resources, and each of these resources may have further exclusion requirements, creating a complex web of interdependencies. In such situations, the use of synchronization contracts protects the server against race conditions and other safety violations, even when users add their own modules to the system. When a deadlock cannot be automatically avoided, the contracts of the modules involved in the deadlock can be analyzed to help locate the cause of the problem and resolve the deadlock.

1.3 Semantics of the contract model

To date, we have extended two object-oriented languages—Ruby and Eiffel—to support our contract model, and we believe that similar extensions could apply to most strongly typed object-oriented languages. The main contribution of this paper is the formal semantics of our contract model as a language-neutral language extension. We do not define an operational semantics, as our primary objective is not to perform concurrency analysis on specific programs; rather, it is to demonstrate how concurrency constraints and the run-time machinery required to implement them compose and interact with the core functionality of a program that is assembled from contract-aware modules. The key to defining the semantics in this manner is to express the specification in terms of multiple views which compose by conjunction [29]. Using the Z notation [47], we develop a base-language view, which generalizes most of the major strongly-typed object-oriented languages. We then define features of synchronization contracts as separate views, which compose with the base-language view by conjunction. The resulting semantics clearly defines the meaning of concurrency constraints and does so as a clean extension of the base-language semantics.

---

2The extended Eiffel compiler is available at: [http://www.cse.msu.edu/~behrends/universe](http://www.cse.msu.edu/~behrends/universe)
On another level, this manner of defining the semantics of a contract model as an extension to an existing language may also be useful for other contracting models and for comparing two competing models. Generally speaking, when an existing programming language is extended to support a contracting model, the result is a multi-paradigm language, which expresses different concerns using language features that appeal to different programming paradigms [54]. Whereas synchronization contracts enable programmers to express mutual exclusion concerns by writing declarative specifications, an object-oriented programming language that has been extended with synchronization contracts is necessarily a multi-paradigm language. Multi-paradigm programming requires that, in addition to understanding the code for each concern in isolation, the programmer understands how the code for the individual concerns composes to form a program. To facilitate this understanding, Zave advocates the use of partial specifications that compose by conjunction [54]. Our view-based semantics of concurrency constraints fits this bill, and we believe that view-based approaches might also be useful for defining the semantics of other contracting models.

The remainder of the paper is structured as follows. We begin by surveying prior work in language extensions for concurrency (Section 2). We then describe concurrency constraints and present examples to provide intuition that will help in following the formal development and to illustrate advantages of our extension (Section 3). Next, we develop a general model of object-oriented languages and show how concurrency constraints are integrated into a language’s type system (Section 4). The heart of our specification describes how operations in the base language affect synchronization units, realms, and process schedules at run-time. To cleanly separate concerns, we specify base-language, unit, realm, and scheduling semantics as separate views that compose by conjunction (Section 5). When composing these views, we use a deductive proof theory to define the semantics of constraints (Section ??). We conclude with a brief discussion of related work, some novel aspects of our approach, and directions for future research (Section 8). The appendix provides a summary of the Z notation used in the paper (Appendix A). All of the Z definitions in this paper have been type checked using the Z/EVES prover [45].

2 Prior work in language extensions for concurrency

Historically, the approaches to incorporating concurrency features into programming languages have clustered around two fundamental mechanisms for implementing inter-process communication—reading and writing shared storage and explicit message passing. In the shared storage approach, a system is viewed as a collection of concurrent threads that communicate by reading and writing objects in a shared memory (Section 2.1). In the message passing approach, a system is viewed as a collection of concurrent processes that communicate by passing messages over explicit channels (Section 2.2).

2.1 Shared storage approaches

The shared storage approach treats a program as the concurrent composition of sequential threads, which communicate by reading and writing objects in a shared memory. When programming under this approach, the key safety concern
is to avoid race conditions. Following the terminology of Netzer and Miller [39], we distinguish two types of races. A *data race* occurs when there is a concurrent, uncoordinated access to the same memory location. The classic example of a data race occurs when the execution of machine instructions to read a value at some memory location is interleaved with the execution of instructions by a different thread to write a value to that same location. Data races can be avoided by using single-assignment variables, such as those provided by STRAND [42] or Compositional C++ [16], or by requiring all readers and writers to acquire and release locks prior to reading or writing to a location in shared memory.

By contrast, a *general race* occurs when sequences of events by multiple threads execute non-deterministically, in a way that might change the result of a computation. A classic example of a general race occurs in statements such as:

\[
\text{if supplier.condition()} \text{ then } \text{supplier.action()} \text{ end} \quad (1)
\]

where we assume all methods of supplier execute atomically. Because these methods execute atomically, there can be no data races on supplier. However, if two threads \(T_1\) and \(T_2\) are executing (1) concurrently, there exist interleavings in which \(T_2\) can invalidate supplier.condition() after \(T_1\) has checked the condition but before \(T_1\) has executed supplier.action().

Shared storage approaches tend to assign responsibility for avoiding races to either the objects being shared (the suppliers) or the objects performing the shared accesses (the clients). Supplier-side synchronization mechanisms—e.g., *synchronized* methods in Java and *protected* objects in Ada—trace back to monitors [23]; whereas client-side mechanisms—e.g., *synchronized* blocks in Java, *lock* statements in Sather [19], *holdif* statements in CEE [31], procedures with *separate arguments* in SCOOP, and non-blocking transactions in Java [22]—trace back to *conditional critical regions* [21]. Supplier-side mechanisms have the advantage that they easily protect against data races. For example, if supplier were implemented as a monitor, then all accesses to its methods would be mutually exclusive. Providing similar protection using a client-side mechanism requires that all clients bracket accesses to supplier with code that acquires and then releases a lock [9].

On the other hand, supplier-side synchronization mechanisms cannot coordinate interactions that involve multiple suppliers, and they do not naturally support the avoidance of general races. For example, if a client needs atomic access to multiple monitors, the programmer must encapsulate the accesses into a separate service in one of the monitors for the client to invoke. Similarly, to protect against general races such as illustrated by (1) using monitors, the programmer must encapsulate such sequences into a separate service in the supplier. Unfortunately, these solutions require that the supplier developer anticipates all such transactions and leads to a proliferation of services that clutter the supplier interface.

More generally, supplier-side synchronization requires that the supplier maintains history information, such as what parts of a transaction have been completed, which can greatly complicate the supplier’s code [9, 25, p. 85]. In contrast, when the client is responsible for synchronization, the transaction history does not have to be replicated in the supplier, as the client knows implicitly where it is in a complex transaction. In addition to complicating program logic, the need to maintain history information can lead to supplier-side inheritance anomalies [33, 25, p. 52].
Concurrency constraints are declared in client modules and are, therefore, client-side contracts. As such, they can specify transactions involving multiple suppliers. In addition, because realms are always disjoint, our run-time system protects against data races [6]. Thus, concurrency constraints provide the main benefit of a supplier-side mechanism while avoiding most of the pitfalls.

### 2.2 Messaging approaches

The messaging approach treats a program as the concurrent composition of sequential processes, each of which can access only memory local to that process. Processes communicate by passing messages across explicit channels. Within this general approach, we distinguish approaches by the synchrony or asynchrony of messaging. Under synchronous messaging, the sender of a message blocks until the recipient is ready to receive the message, at which point the sender and recipient are said to rendezvous. Many formal notations, e.g., CSP [24], CCS [36], and LOTOS [10], and some programming languages, e.g., Ada [51] and Occam [27], support rendezvous synchronization. While elegant and powerful, rendezvous synchronization has proved difficult to integrate into object-oriented languages.

Under asynchronous messaging, the sender of a message does not block while waiting for the recipient to receive the message. Object-oriented languages that support asynchronous messaging define a so-called active object [2, 40], which queues client requests for services and processes the queued requests in a local event loop.³ Carmel and colleagues developed an active-objects model that makes the asynchrony of messaging transparent using a principle called wait by necessity [14]. Under this principle, a process that invokes a value-returning method of an active object continues executing, but blocks when it tries to use the return value if the asynchronous invocation has yet to complete. Active object extensions apply to a wide range of object-oriented languages and integrate cleanly with existing styles of object-oriented programming. To date, active objects have been integrated into Eiffel [14], C++ [3], and Java [15].

The messaging approach has several virtues. First, because processes communicate only through messages, programs written using this approach can never exhibit data races. Second, there exist powerful notations and calculi for reasoning about the behavior of programs in which processes communicate by passing messages. The operational nature of these notations enables simulation of program executions and exhaustive analysis to uncover faults that are difficult to produce during testing. The ability to analyze a message passing program for temporal properties is a major benefit over approaches based on shared-storage.

On the other hand, messaging is inherently a supplier-side synchronization mechanism. Thus, programs developed using this approach are subject to general races and history sensitivity [25, 9]. To address these concerns, a process can maintain a state-machine representation that prescribes the legal sequences of messages and that it consults at runtime when deciding whether to accept a message (e.g., Procol [52] and Nierstrasz’s Regular Types [40]). This solution requires that a process’ state machine describes all possible ways in which clients can legally send it messages.

Some problems are most naturally modeled by multiple processes operating on shared data. Solutions to such problems under the messaging approach typically require a new process for each shared object, and so incur perfor-

---

³Such an event loop is often called a body. For a good discussion of the design and implementation of active objects, see [12].
formance penalties. The Ada-95 rationale cites this problem as one of the reasons for introducing protected types [28, II.9].

In summary, the shared-storage and message-passing approaches lead to different synchronization problems. Because invoking a method of a shared object is not atomic, the main problem for a program with shared objects is that of regulating access to the shared objects. On the other hand, because unanticipated interleavings of asynchronous atomic events can produce unintended causality relationships [46], the main problem for a program that uses active objects and messaging is that of coordinating these events to preserve causality and compute global properties of multiple active objects.

3 Concurrency constraints

Our contract model provides for client-side concurrency management in the shared storage model. This section describes the key concepts and definitions required to understand the model (Section 3.1). We provide two examples to illustrate the kinds of synchronization problems that our model addresses. To be concrete, we couch all examples in our Eiffel extension. The examples were chosen to be familiar, but also to showcase some of the more subtle features of the model, namely initialization semantics (Section 3.2) and conditional unit references (Section 3.3). We also use examples to illustrate how realms grow and shrink during execution (Section 3.4) and how composition of synchronization contracts addresses a well-known problem of contract models for object oriented languages (Section 3.5).

3.1 Concepts and definitions

A process is a thread of control in a program. Its data space, or realm, comprises a collection of objects. These objects may not be legally accessed by other processes; we therefore say that a process owns the objects in its realm. Over the lifetime of a program, an object might change owners many times, but it is never owned by more than one process at a time. When the owner of an object changes, we say that the object migrates out of the realm of one process and into the realm of another.

Rather than dealing with individual objects, our model partitions objects into synchronization units, with the property that objects in a synchronization unit migrate in and out of realms together. Synchronization units are created when certain synchronization classes contained in a program are instantiated, and destroyed when the instances of these synchronization classes are destroyed. A programmer should declare any class whose objects have bodies that are to execute in processes of their own to be a synchronization class. Other classes may be declared as synchronization classes to provide the programmer with finer control over concurrency.

When instantiating a synchronization class, the run-time system produces a new object o, as usual, but it also creates a new synchronization unit u and installs o as the root object of u. The unit u exists for as long as its root exists.

4Other reasons are the potential for indirect race conditions due to abstraction inversion and the control-oriented nature of rendezvous synchronization, which the designers felt was “out of line with a modern object-oriented approach.”

8
during which time \( u \) provides a container for all objects that are created by objects in \( u \), except for those objects that themselves create synchronization units. Thus, every object in the system belongs to exactly one synchronization unit over the course of its lifetime and each synchronization unit contains at least its root object. In the sequel, when we speak of a program creating or destroying a synchronization unit, we mean that the program is creating or destroying the root object of the synchronization unit.

Because the run-time system migrates the objects in a synchronization unit together, a process \( P \) owns an object in unit \( u \) if and only if \( P \) owns every object in \( u \). Thus a realm comprises a collection of synchronization units, and we say \( P \) owns \( u \) if \( u \) is in the realm of \( P \). In addition, every process owns at least one synchronization unit that never migrates out of its realm. For a process to legally access an object, it must own the object, and thus also own the unit that contains the object. Our contract model prescribes that an illegal access results in a run-time exception.

One goal of our model is to raise the level of abstraction for expressing interprocess synchronization. Instead of writing explicit synchronization code, the programmer provides a declarative specification of a unit’s data needs in the form of a concurrency constraint. A realm is deemed sufficient if the concurrency constraint of each of the synchronization units contained in the realm is satisfied. A realm is deemed minimally sufficient if it is sufficient and it cannot be contracted without violating sufficiency. A process is scheduled for execution if and only if its realm is minimally sufficient.

The concurrency constraint for a synchronization unit describes the conditions under which the unit might access other synchronization units. It is a limited propositional formula in two kinds of variables, called unit variables and condition variables. A unit variable references another synchronization unit, which therefore plays a role as a supplier. In the producer/consumer example, for instance, the producer and consumer declare unit variables that reference the channel, which provides methods that the producer and consumer invoke to put data into and extract data from the channel.

A condition variable encodes, as a boolean value, some aspect of a unit’s state that affects the unit’s synchronization needs. For instance, a condition variable, say putting, in the producer may signify whether the producer is putting data into the channel or not, and a condition variable, say getting, in the consumer may signify whether the consumer is getting data from the channel or not; the channel may similarly provide two condition variables, say full, to signify when the channel is full, and empty, to signify when the channel is empty. A concurrency constraint in the producer can then express that the producer’s putting variable should be true only if the channel is in the producer’s realm and the channel’s full variable is false—in other words, the producer can put data into the channel only when it has exclusive access to the channel and the channel is not full. Similarly, a concurrency constraint in the consumer can express that the consumer can extract data from the channel only when it has exclusive access to the channel and the channel is not empty.

Operations that may modify the values of unit variables or condition variables are called realm affecting operations. When a process executes a realm affecting operation, the run-time system migrates units into and out of the realms of those processes whose realms are not minimally sufficient in an attempt to make them so. It then schedules those processes whose realms are made minimally sufficient and blocks those processes whose realms are not.
realm-affecting operations, the realm of a process is stable, which is to say that no unit in the realm will migrate into another realm. This stability is needed to guarantee mutual exclusion. Deadlock occurs if there is a cycle of processes, each blocked and waiting for the next process in the cycle to release a unit in order to migrate the unit and satisfy a concurrency constraint. Our algorithms that implement dynamic negotiation of realms employ advanced techniques for deadlock and starvation avoidance. (See [6] for details.)

To simplify both the implementation and formalization of our contract model, we limit concurrency constraints to use only unit variables and condition variables and we identify condition variables and boolean variables. We could introduce special syntax to distinguish condition variables from boolean variables, both when they are created and when they are manipulated, thereby highlighting instructions in the body of a module that affect synchronization. Additionally, the meanings of condition variables could be defined declaratively, instead of operationally, to further separate the specification of synchronization from functional code. In the producer/consumer system, for example, the channel might declare the condition variable full using a boolean expression, such as

```
full: condition = ('count == MAX_CAPACITY )
```

where we assume condition is a subtype of bool provided for condition variables, count is a local variable that maintains the number of the data items in the channel, MAX_CAPACITY is a constant denoting the maximum number of data items that the channel can hold, and the tick mark (‘) suppresses evaluation of an expression. Moreover, the level of abstraction of synchronization contracts can be further raised by encoding typestates [49, 17] as condition variables and introducing mechanisms for declaring how events (e.g., entering or returning from a method invocation, reaching a location in a method body, etc.) cause an object to transition from one typestate into another. Since all of these proposals can be defined by translation to the simpler contract model, we restrict our attention in this paper to this simple model. Hereafter, therefore, we consider concurrency constraints that are expressed in terms of just unit variables and condition variables, and we equate condition variables with boolean variables, which are explicitly updated in functional code.

### 3.2 Example: Multi-resource synchronization contracts

Consider a centralized event manager. Upon receiving events dispatched by some event source, a centralized event manager announces the events to listeners, which register interest in the events. Such an event manager might be used, for example, in a program that simulates a physical phenomenon and displays different aspects of the simulation in two separate windows. When a simulation object (event source) makes a change that could affect one of the displays, it dispatches an event to the event manager, which then announces the event to all registered display objects (listeners). If the display objects execute in different threads, concurrency constraints will be needed to ensure that the displays always update in synchrony, or the user might perceive an inconsistency in the two displays. Thus, event dispatching must synchronize with event announcement across all registered listeners.

**Figure 1** depicts such an event manager implemented in our extension to Eiffel. In this example, two display units implement the EVENT_LISTENER interface (whose definition is omitted). Listeners register interest by being passed
synchronization class EVENT_MANAGER -- synchronization class

creation

make

feature { NONE } -- attributes

listener1, listener2: EVENT_LISTENER -- unit variables

feature { ANY } -- creation

make(l1, l2: EVENT_LISTENER) is

  do
  
  listener1 := l1
  
  listener2 := l2
  
  end

feature { ANY } -- client routines

dispatch(an_event: EVENT) is

  do
  
  listener1.announce(an_event)
  
  listener2.announce(an_event)
  
  end

concurrency -- concurrency constraints

listener1 and listener2

end -- class EVENT_MANAGER

Figure 1: Example with null unit referents
to the make procedure (lines 7–11), which is executed upon creation of an event manager. The class contains two unit variables, listener1 and listener2 (line 5), and the associated concurrency constraint is the conjunction of these variables (line 19). We interpret this constraint as asserting that the event manager requires exclusive access to the synchronization units referenced by listener1 and listener2 in order to execute. This constraint prevents a process from owning an event manager unless it also owns both listeners. Thus, if a process requires exclusive access to an event manager, e.g., when executing within a unit that needs to dispatch an event, the process will block until the event manager and both listener units can be migrated into its realm.

This concurrency constraint poses an interesting semantic problem: How should an uninitialized unit variable be interpreted when evaluating a constraint? This problem arises, for example, when executing the make procedure, which first initializes listener1 (line 9) and then listener2 (line 10). Line (9) contains a realm-affecting operation. Thus, its execution causes the concurrency constraint to be evaluated. At this point, listener1 references an object of class EVENT LISTENER, and listener2 contains the value null. A unit reference evaluates to true if the referent is in the current realm and false if the referent is not in the current realm, but how should we interpret a unit reference that has no referent? Under the false interpretation, the example constraint will evaluate to false, and the process running this code will block indefinitely, despite there being no possible sharing of objects among processes. Interpreting a null reference as true instead leads to the behavior that a programmer would expect; namely, that unit variables cannot affect the concurrency constraint until after they have been initialized.

In summary, we interpret a null reference as true to allow programmers to write non-deadlocking initialization code in the traditional object-oriented style. Under this interpretation, a null unit variable does not prevent a process from executing. If a process dereferences a null unit variable, then the run-time system generates a null-pointer exception; an exception is more observable, and thus more amenable to being found and fixed, than a locked-out process. Thorny semantic issues, such as this one, arise in virtually all languages that employ features from multiple paradigms. Because features from different paradigms are seldom entirely orthogonal, their interactions must be carefully considered and defined. Our formal semantic model helps raise and clarify these issues.

3.3 Example: Conditional synchronization contracts

Figure 2 depicts an implementation of a CHANNEL MONITOR class, which watches for events over a channel and writes high priority events on display. A CHANNEL MONITOR unit consumes data supplied over a channel, where CHANNEL is a synchronization class. (The definition of CHANNEL is omitted.) The concurrency constraint (line 29) predicates the conditional unit reference, channel when channel.hasData, on reading being true. This concurrency constraint enforces the following behavior: While reading is false, nothing is required; however, if a process assigns reading the value true and if the referent of channel is non-null, the process blocks unless it owns the referent and the referent’s condition variable hasData is true. Thus, this constraint ensures more than mutual exclusion; it also defers the migration of a channel unit until the channel contains data. We call channel.hasData a remote condition variable because it refers to a condition variable of a different synchronization unit, the referent of
This example also illustrates the distinction between synchronization units and normal objects. When we instantiate class \texttt{CHANNEL\_MONITOR}, several things happen. In addition to creating a new \texttt{CHANNEL\_MONITOR} object \( o \), the system also creates a new unit \( u \) to contain \( o \). However, when the system creates a \texttt{DISPLAY} object (line 12), it does not create a new unit, but simply adds the new object to \( u \).

### 3.4 Example: Realm dynamics

Figure 3 shows a series of UML instance diagrams representing global states at different stages during execution of an example program. The stages show how realms grow and shrink in order to satisfy the concurrency constraints of the various synchronization units. The example program creates three processes: one \texttt{SOURCE} process and two \texttt{CHANNEL\_MONITOR} processes. The corresponding realms are depicted using dashed or dotted boxes that encompass the synchronization units contained in each realm. A realm depicted using a dashed (resp. dotted) box indicates that the corresponding process is enabled (resp. blocked). These processes operate over a total of six synchronization units—one \texttt{SOURCE} unit, one \texttt{EVENT\_MANAGER} unit, two \texttt{CHANNEL} units, and two \texttt{CHANNEL\_MONITOR} units—each indicated by its root object. The source object sends events to the event manager, which broadcasts them to the two channels, from which they are read by the respective channel monitors.

Unit variables are indicated as labels on references (arrows) connecting synchronization units. For example, the \texttt{SOURCE} unit declares one unit variable, \texttt{manager}, which references the \texttt{EVENT\_MANAGER} unit. Condition variables are boolean valued attributes. Thus, the \texttt{SOURCE} unit declares one condition variable, \texttt{hasEvt}. Intuitively, \texttt{hasEvt} is true when source has an event that it can transmit; and false, otherwise. Concurrency constraints are shown in curly braces below or alongside the respective synchronization units. For instance, the concurrency constraint of the \texttt{SOURCE} unit is \texttt{hasEvt} \( \Rightarrow \) \texttt{manager}, signifying that a process executing this unit needs exclusive access to the manager when it has an event to transmit. The \texttt{EVENT\_MANAGER} unit receives events and broadcasts them to the two \texttt{CHANNEL} units, as described in Section 3.2. The \texttt{CHANNEL\_MONITOR} units follow the description in Section 3.3, reading events from the channels. Neither of the channels has a concurrency constraint.

Initially, the realms of all processes contain just the root instances of their respective synchronization units, as shown in Stage 0 of Figure 3. In addition, the \texttt{hasEvt} variable is false, indicating that source does not have an event to send; both \texttt{hasData} variables are false, indicating that the channels do not contain data; and both \texttt{reading} variables are true, indicating that the channel monitors need to read data from their respective channels. The concurrency constraint of source is trivially satisfied, and so its realm is complete; whereas the constraints of the event monitors are not satisfied, and so their realms are not complete. Moreover, the realms of the event monitors cannot be completed by migrating synchronization units—the respective channels must first assign \texttt{true} to \texttt{hasData}. Thus, the processes that own the \texttt{CHANNEL\_MONITOR} units are blocked.

Stage 1 of Figure 3 begins when the source generates an event and assigns \texttt{hasEvt} the value \texttt{true}. Its realm expands to include the event manager, in order to satisfy the constraint associated with the source, and both channels,
(1) synchronization class CHANNEL_MONITOR

(2) creation

(3) make

(4) feature { NONE } -- private attributes

(5) channel: CHANNEL -- a unit variable since CHANNEL is
   -- declared to be a synchronization class
   -- CHANNEL declares hasData

(6) reading: BOOLEAN -- a condition variable

(7) display: DISPLAY -- not a synchronization class

(8) feature { ANY } -- creation

(9) make(a:channel: CHANNEL) is

(10) do

(11)    channel := a:channel

(12)    create display

(13)    reading := false

(14) end

(15) feature { ANY } -- process entry point

(16) start is

(17) local event: EVENT

(18) do

(19) from until false loop -- infinite loop

(20)    reading := true

(21)    event := channel.pop

(22)    reading := false

(23) if event.priority >= 100 then

(24)       display.show(event)

(25) end

(26) end

(27) end

(28) concurrency

(29) reading => channel when channel.hasData

(30) end -- class CHANNEL_MONITOR

Figure 2: Example of a conditional unit reference
in order to satisfy the constraints associated with the event managers. In Stage 2, after the event manager forwards the event to both channels, the channels assign true to their hasData variables and the source assigns false to its hasEvt variable. Because the concurrency constraints now indicate that the source no longer needs the EVENT_MANAGER or CHANNEL units, its realm is contracted. Simultaneous with contraction of the source realm, the realms of the two channel monitor processes expand to include their respective channels, completing their realms and allowing the processes to read the data. Finally, in Stage 3, the channels are empty and, as the channel monitors are no longer reading data from the channels, their realms are contracted.

3.5 Example: Implicit composition of synchronization contracts

The previous example illustrates how the concurrency constraint of the source, a local contract between the source and the event manager, has farther reaching effects: in Stage 1 of Figure 3, when expanding the realm of the source to
include the event manager, the concurrency constraint of the event manager must also be satisfied; the realm is therefore expanded to also include the channels. In this section, we show how this implicit composition of synchronization contracts addresses a well-known problem that arises using layered architectures.

The layering problem is illustrated in Figure 4 by the collaboration between the clients, $C_1$ and $C_2$, the suppliers, $S_1$ and $S_2$, and the delegate, $D$. Here, the clients form a top layer, the suppliers form a middle layer, and the delegate forms a bottom layer. Because the delegate is shared by the two suppliers, a transaction in either client may require exclusive access to the delegate. However, the client does not know how the supplier is implemented, and therefore cannot know about the delegate. This problem is solved elegantly by composing synchronization contracts. For example, when a process executing $C_1$ (resp. $C_2$) assigns $T$ the value true, indicating that it needs to perform a transaction, the run-time system will block further execution of the process until the realm contains both $S_1$ (resp. $S_2$) and $D$. Thus, the clients safely access the delegate indirectly through their suppliers without needing to know how the suppliers implement their services.

Now suppose that one of the suppliers is replaced with a new supplier that does not access the delegate. Then the concurrency constraint of the new supplier will not reference $D$ and the run-time system will no longer serialize executions of the transactions by the clients. There is no need to modify the implementations of the clients or of the remaining supplier. Thus, this solution preserves modularity and enhances reuse.

In summary, concurrency constraints ensure the integrity of shared resources in systems that implement complex collaborations by propagating local synchronization contracts. However, implicit propagation of local contracts can also produce undesirable non-local effects, such as deadlock. Cycles in the client-supplier relation create the potential for deadlock regardless of the mechanism used to achieve synchronization. An advantage of our model over region-based synchronization models is that information needed to reason about potential deadlock is localized in concurrency constraints. We believe that this property of the model makes synchronization relations easier to discern and helps the user better understand how deadlocks might occur. Additionally, it may permit the development of compositional static analysis algorithms to efficiently determine that a larger class of programs is free from deadlock.

4 Type-system extensions for contract awareness

Our contract model extends the type system of a base language with two new features: synchronization classes and concurrency constraints. To formalize this extension, we first develop a general model of object-oriented type systems. Because we are concerned only with base languages for which types are static throughout the lifetime of a program, we define this base-language type system using axiomatic descriptions in $Z$ (Section 4.1). We then extend this type system with the ability to distinguish synchronization classes from non-synchronization classes and to associate concurrency constraints with synchronization classes. Concurrency constraints refer exclusively to condition and unit variables, which are special attributes declared in synchronization classes (Section 4.2). We formally define the constraint language (Section 4.3) and integrate it into this base-language type system by providing a type theory for judging the soundness of a constraint with respect to the synchronization class that declares it (Section 4.4). For brevity, we do
not include the full type theory. The omitted details can be found in [48].

### 4.1 Base-language type system

Any object-oriented program provides classes with named attributes, which we model with two given sets:

\[
[\text{CLASS}_\text{ID}, \text{ATTRIBUTE}_\text{ID}]
\]

These sets comprise *identifiers* that model class and attribute names respectively.\(^5\) Identifiers can only be compared for equality; we use them to define functions that record the association of attributes to the class in which they are declared and the association of attributes to types.

To model the meaning of a class attribute requires a space of legal *types*, which we model using a free type in \(Z\):\(^6\)

\[
\text{TYPES} ::= \text{bool} \mid \text{nat} \mid \text{ref}\langle \text{CLASS}_\text{ID} \rangle
\]

In this simplified model, a type may be either a boolean (\text{bool}), a natural number (\text{nat}), or a reference (\text{ref}) to an instance of some given class. Each class\text{atts} attribute associates to a type in this space. We formalize this notion using two (partial\(^7\)) functions, \(\text{attrType}\), which maps an \(\text{ATTRIBUTE}_\text{ID}\) to its declared type in \(\text{TYPES}\), and \(\text{attrOfClass}\), which maps an \(\text{ATTRIBUTE}_\text{ID}\) to the \(\text{CLASS}_\text{ID}\) in which it is declared.

\[
\begin{align*}
\text{attrType} & : \text{ATTRIBUTE}_\text{ID} \rightarrow \text{TYPES} \\
\text{attrOfClass} & : \text{ATTRIBUTE}_\text{ID} \rightarrow \text{CLASS}_\text{ID}
\end{align*}
\]

\[
\text{dom}(\text{attrType}) = \text{dom}(\text{attrOfClass})
\]

The invariant says that every attribute that has a declared type is declared in some class, and vice versa.

Because synchronization contracts may refer to attributes that reference instances of another class (i.e., \text{ref}-types), we will often talk about the *class referenced by* a given attribute. We formalize this notion as follows:

\[
\begin{align*}
\text{attrRefsClass} & : \text{ATTRIBUTE}_\text{ID} \rightarrow \text{CLASS}_\text{ID} \\
\text{attrRefsClass} & = \text{attrType} \circ \text{ref}^\sim
\end{align*}
\]

The right-hand side of the invariant in this definition composes \(\text{attrType}\), which looks up an attribute’s type, with \(\text{ref}^\sim\), which maps types that reference a \(\text{CLASS}_\text{ID}\) to said \(\text{CLASS}_\text{ID}\). These three functions, \(\text{attrType}\), \(\text{attrOfClass}\), and \(\text{attrRefsClass}\), describe the type system of an object-oriented language in sufficient detail for extension with our model of synchronization contracts.

### 4.2 Unit classes and unit and condition variables

\(^5\)The only difference being that identifiers are unique, whereas names need not be unique.

\(^6\)Use of the word “type” in the latter context, to refer to a meta-language type, should not be confused with its use in the former context, to refer to a type in a program. When defining semantics of a programming language, such overloading of terminology in describing the language and the meta-language is commonplace.

\(^7\)Henceforth, all functions are considered to be partial unless explicitly qualified as being total.
To integrate our model into a given base language requires extending that language’s type system with the ability to identify synchronization classes and process classes (whose instances should run in their own thread), to associate concurrency constraints with synchronization classes, and to distinguish certain class attributes as condition variables and unit variables. The set of synchronization classes is a subset of the classes that are declared in a program, and the set of process classes is a subset of the synchronization classes:

\[ \text{SynchClasses} : \mathbb{P} \text{CLASS} \]
\[ \text{ProcessClasses} : \mathbb{P} \text{CLASS} \]
\[ \text{ProcessClasses} \subseteq \text{SynchClasses} \]

Condition and unit variables are then just distinguished attributes of synchronization classes.

\[ \text{UnitVars} : \mathbb{P} \text{ATTRIBUTE} \]
\[ \text{ConditionVars} : \mathbb{P} \text{ATTRIBUTE} \]
\[ \text{UnitVars} = \text{dom}(\text{attrRefsClass} \supset \text{SynchClasses}) \cap \text{dom}(\text{attrOfClass} \supset \text{SynchClasses}) \]
\[ \text{ConditionVars} = \text{dom}(\text{attrType} \supset \{\text{bool}\}) \cap \text{dom}(\text{attrOfClass} \supset \text{SynchClasses}) \]

The invariant says that a unit variable is a reference to an instance of a synchronization class that is itself an attribute of a synchronization class, and a condition variable is a \text{bool} attribute of a synchronization class.

In the sequel, we speak of the unit or condition variables of a class. An attribute is a unit variable of class \( C \) if it is both a unit variable and an attribute of \( C \). For example, \text{channel} is a unit variable of \text{CHANNEL\_MONITOR} (Figure 2). We distinguish two different kinds of condition variables of a class: An attribute \( a \) is a local condition variable of class \( C \) if it is both a condition variable and an attribute of \( C \), and it is a remote condition variable of \( C \) if there exists an attribute \( u \), such that \( u \) is a unit variable of \( C \) and \( a \) is a condition variable of the (synchronization) class referenced by \( u \). For example, \text{reading} is a local condition variable of \text{CHANNEL\_MONITOR} and \text{hasData} is a remote condition variable of \text{CHANNEL\_MONITOR} (Figure 2). The following definitions formalize these concepts:

\[ \text{uvarOfClass} : \text{ATTRIBUTE} \rightarrow \text{CLASS} \]
\[ \text{cvarOfClass} : \text{ATTRIBUTE} \rightarrow \text{CLASS} \]
\[ \text{remoteCvarOfClass} : \text{ATTRIBUTE} \leftrightarrow \text{CLASS} \]
\[ \text{uvarOfClass} = \text{UnitVars} \triangleleft \text{attrOfClass} \]
\[ \text{cvarOfClass} = \text{ConditionVars} \triangleleft \text{attrOfClass} \]
\[ \text{remoteCvarOfClass} = \text{cvarOfClass} \triangleleft \text{attrRefsClass} \triangleleft \text{uvarOfClass} \]

Observe that an attribute of a non-synchronization class can reference an instance of a synchronization class, but such an attribute is not a unit variable and it cannot appear in a concurrency constraint. Consequently, the programmer

---

8 In our Eiffel extension, the programmer distinguishes a synchronization class by adorning its declaration with the keyword \text{synchronization}. By contrast, the programmer declares that each instance of a specified synchronization class should run in its own thread by declaring the synchronization class to inherit from class \text{PROCESS\_BASE}. Other language extensions may specify this in a different manner. Instantiating such a class produces a new object, a new unit for the new object, a new process, and a new realm for the new process.

9 where the class is assumed to be a synchronization class.

10 and a local condition variable of \text{CHANNEL}; for brevity, we omit the definition of \text{CHANNEL}.
(and not the run-time system) is responsible for ensuring that a process accesses this attribute only when the referent is in the current realm. If the programmer fails in this responsibility, then the synchronization class violates its responsibilities under the contract. Such a violation will automatically generate a run-time exception. The treatment of attributes of a non-synchronization class that reference instances of a synchronization class is one of the issues raised by our semantic model that, previously, we had not explicitly considered or defined.

4.3 Concurrency constraint language

We now define a language for expressing concurrency constraints. A condition is a full propositional formula in which each atomic proposition refers to a condition variable or to both a unit variable and a condition variable.

\[
\text{Condition} ::= \text{condLocal}(\text{ConditionVars}) \\
| \text{condRemote}(\text{UnitVars} \times \text{ConditionVars}) \\
| \text{neg}(\text{Condition}) \\
| \text{and}(\text{Condition} \times \text{Condition}) \\
| \text{or}(\text{Condition} \times \text{Condition}) \\
| \text{implies}(\text{Condition} \times \text{Condition})
\]

The syntax distinguishes a local atomic condition (condLocal) from a remote atomic condition (condRemote), with the former requiring just a condition variable and the latter requiring both a unit variable and a condition variable. The remaining operators designate logical connectives. The constraint in line 29 of Figure 2 illustrates the two types of atomic propositions: reading and channel.hasData denote local and remote atomic conditions, respectively.

In the sequel, we refer to a condition whose atomic conditions are all local as a local condition. More formally:

\[
\text{LocalCondition} := \text{P Condition} \\
\text{LocalCondition} = \bigcap \{X : \text{P Condition} \mid \text{ran}(\text{condLocal}) \subseteq X \land \\
\text{neg}(\{X\}) \subseteq X \land \text{and}(\{X \times X\}) \subseteq X \land \\
\text{or}(\{X \times X\}) \subseteq X \land \text{implies}(\{X \times X\}) \subseteq X\}
\]

The invariant says that LocalCondition is the least set X that contains the set of local atomic conditions (ran(condLocal)) and all conditions formed by applying logical connectives to elements of X.

A concurrency constraint is a limited propositional formula in which each atomic proposition refers to no variables, a single unit variable, or both a single unit variable and a single condition variable.

\[
\text{Constraint} ::= \text{true} \\
| \text{unitvar}(\text{UnitVars}) \\
| \text{when}(\text{UnitVars} \times \text{Condition}) \\
| \text{IMPLIES}(\text{LocalCondition} \times \text{Constraint}) \\
| \text{AND}(\text{Constraint} \times \text{Constraint})
\]

The first operator (true) denotes the constraint that always evaluates to true. There are two kinds of unit references—simple (unitvar) and conditional (when)—both of which indicate a data-access dependence on another unit. A simple unit reference asserts that the referent is in the same realm as the unit to which the constraint applies. In contrast, a conditional unit reference asserts that the referent is in the same realm and an additional referent condition holds. For
example, in line 19 of Figure 1, listener1 is a simple unit reference; whereas, in line 29 of Figure 2, channel when channel.hasData is a conditional unit reference. The latter asserts that the CHANNEL unit referenced by channel is in the realm and, moreover, that the hasData variable of this unit must be true when the unit is migrated into the realm. The implication operator (IMPLIES) denotes a constraint that is guarded by a local condition, and the and operator (AND) denotes the logical conjunction of two constraints. Our constraint logic is not complete; it lacks negation and disjunction and provides a limited form of implication, where the guard must be a local condition. This design represents a balance between expressiveness and simplicity, allowing efficient checking of constraint satisfaction [5].

4.4 Integration

To fully integrate concurrency constraints into a type system, we must associate them with synchronization classes and show how to check them for type correctness. Each synchronization class has a unique concurrency constraint, which specifies the conditions under which instances of the class may access other synchronization units. Here, and in the sequel, we use the term associated class to mean the class with which a given concurrency constraint is associated. Equivalently, we say that a concurrency constraint applies to the class with which it is associated. We model this association using a function called constraint, which maps a unit class to its constraint. To properly define this function requires formalizing the type correctness of a constraint with respect to its associated class.

To define type correctness, we provide a type theory for conditions and constraints. A type theory is a deductive system of axioms and inference rules that conclude typing judgements [20]. Formally, a typing judgement is a triple of the form \( C \vdash E \) where \( C \) names a synchronization class, \( \vdash \) names a judgement relation (of which there are two in our theory), and \( E \) is an expression whose type correctness is being judged with respect to \( C \). For example, the typing judgement \( C \vdash \text{cond} \ T \) asserts, “when associated with class \( C \), condition \( T \) is well typed.” Our type theory defines two type relations:

\[
\vdash \text{cond} : \text{CLASSJD} \leftrightarrow \text{Condition}
\]

\[
\vdash \text{concur} : \text{CLASSJD} \leftrightarrow \text{Constraint}
\]

where \( \vdash \text{cond} \) is used to define \( \vdash \text{concur} \).

Type theories are defined by induction over the structure of the expressions being judged—in this case, conditions and constraints. Following standard conventions, rules in our type theory take the general form:

\[
P, B \vdash Q \quad \text{[rname]}
\]

Here, \([\text{rname}]\) provides a name by which to designate the rule; \( Q \) is a typing judgement, called the conclusion; \( P \) is a (possibly empty) set of typing judgements, called premises; and \( B \) is a boolean expression, called the side condition. The rule is read as saying “if the typing judgements in \( P \) are true and if \( B \) is true, then the typing judgement \( Q \) is also true.” An axiom is a rule in which \( P \) is empty, allowing \( Q \) to be inferred provided that the side condition is true;

11The when-clause is a new syntax added after the publication of [4]. It replaces: reading \( \Rightarrow \) queue.has\_data \( \land \) queue with reading \( \Rightarrow \) queue when queue.has\_data
whereas an inference rule is one in which \( P \) contains one or more typing judgements, embodying proof obligations that must be discharged to infer \( Q \).

We judge a condition \( cnd \) to be well typed when associated with class \( C \), written \( C \vdash_{\text{cond}} cnd \), if (1) each local atomic condition in \( cnd \) refers to a condition variable in \( C \), and (2) each remote atomic condition in \( cnd \) refers to a unit variable \( u \) in \( C \) and a condition variable of the class that \( u \) references. Table 1 depicts the axioms in this type theory for conditions. Axioms \([\text{lcLocalVar}]\) and \([\text{lcRemoteVar}]\) correspond to, respectively, case (1) and case (2). For example, the following typing judgement:

\[
CHANNEL\_MONITOR \vdash_{\text{cond}} \text{condRemote}(\text{channel}, \text{hasData})
\]

asserts \( \text{channel.hasData} \) is a well-typed condition for class \( CHANNEL\_MONITOR \). We could not conclude:

\[
CHANNEL\_MONITOR \vdash_{\text{cond}} \text{condLocal}(\text{listener1})
\]

because class \( CHANNEL\_MONITOR \) does not declare a boolean variable named \( \text{listener1} \). Other rules in this type theory allow judgements involving conditions that use logical connectives. For brevity, we omit these rules, which are standard and can be found in [48].

Table 2 lists some of the rules that define the \( \vdash_{\text{concur}} \) relation. The rule \([\text{unitRef-1}]\) asserts that a simple unit reference names a unit variable of the given class. The rule \([\text{unitRef-2}]\) asserts that a conditional unit reference names a unit variable of the given class and a condition that is well-typed with respect to the given class. Moreover, every remote condition variable that this condition refers to must be local to the named unit variable. To enforce this restriction, we make use of the \( \text{referentCond} \) relation, which relates, to each unit variable, the set of conditions whose
remote atomic conditions refer to only that unit variable. This relation is formalized as follows:

\[
\begin{aligned}
\text{referentCond} : \text{ATTRIBUTE JD} & \rightarrow \mathcal{P} \text{ Condition} \\
\text{dom}(\text{referentCond}) &= \text{UnitVars} \\
\forall u : \text{UnitVars} & \bullet \\
\text{referentCond}(u) &= \\
\cap \{ X : \mathcal{P} \text{ Condition} \mid \\
&\quad \text{ran}(\text{condLocal}) \subseteq X \land \\
&\quad \text{ran}(\{u\} \times \text{ConditionVars}) \subseteq \text{condRemote} \subseteq X \land \\
&\quad \text{reg}[X] \subseteq X \land \text{and}[X \times X] \subseteq X \land \\
&\quad \text{or}[X \times X] \subseteq X \land \text{implies}[X 	imes X] \subseteq X \}
\end{aligned}
\]

This definition generates, for each unit variable \(u\), the set of all syntactically legal conditions in which all remote atomic conditions refer to \(u\). For example, \text{channel}.\text{hasData} belongs to \text{referentCond}(\text{channel}) from which we infer that the concurrency constraint in line 29 of Figure 2 is type correct. Table 2 also lists the rule for \text{IMPLIES}, which requires that the condition and the constraint are type correct with respect to the associated class, and the rule for \text{AND}.

We may now formally define the relationship between a unit class and the concurrency constraint associated with the class:

\[
\begin{aligned}
\text{constraint} : \text{CLASS JD} & \rightarrow \text{Constraint} \\
\text{dom}(\text{constraint}) &= \text{SynchClasses} \\
\forall C : \text{dom}(\text{constraint}) & \bullet C \vdash_{\text{concur}} \text{constraint}(C)
\end{aligned}
\]

This axiomatic description extends the base-language type system with support for concurrency constraints.

To recap, we first extended the generic base-language type system with sets that denote synchronization classes, unit variables and condition variables. We then defined the constraint language in terms of these new sets and demonstrated how to judge the type correctness of a constraint with respect to its associated class. With the addition of the \text{constraint} function, which associates concurrency constraints with synchronization classes, the extension is complete. This model captures the integration of synchronization classes and concurrency constraints into a large set of object-oriented base languages, requiring only that the extended language provide some syntactic mechanism for distinguishing synchronization classes and unit/condition variables, and for associating concurrency constraints with synchronization classes.

5 Two-tier model of contract-aware programs

We developed a two-tier model of the semantics of programs written in a language that has been extended with synchronization contracts. The first tier models base-language programs, units, and realms, and the causal connection between operations in a base-language program and the creation/deletion of units and realms, with no concern for contract negotiation or process synchronization, which are modeled in the second tier. Recall that synchronization units and realms are not native to a base language, and to minimize the intrusiveness, we opted to not provide primitives for
explicitly manipulating them. Instead, units and realms are created, updated, and destroyed as a side-effect of executing ordinary base-language instructions, and processes are scheduled so as not to violate any concurrency constraints. This section introduces three distinct models—of base-language programs, programs that manipulate units, and programs that manipulate realms—which we specify as views [29] that can be composed to model the causal connection between base-level operations in a contract-aware program and operations on units and realms. The resulting model is called the free program model to suggest that programs in this model are free of synchronization constraints. In addition to developing the free program model, this section also describes a selection of predicates and operations on this model that will be used in modeling the second tier (Section 7).

5.1 Base-language view

Our model of the base-language view borrows ideas and terminology from Woodcock and Davies [53, Ch. 16], who formalize the execution of a program as a sequence of operations over a global state space. In our model, a base-language program $P$ denotes a state space $\mathcal{P} = (\text{BaseView}, P\text{Init}, \{i : I \bullet P\text{Op}_i\})$, where BaseView is a Z schema that represents objects and object attributes, $P\text{Init}$ is an initialization schema, and the indexed set is the set of operations that represent steps of program $P$. Each operation $(P\text{Op}_i)$ is a schema of the form $\Delta\text{BaseView}$. Then an execution of the program is a sequence:

$$P\text{Init} \Downarrow P\text{Op}_\alpha \Downarrow P\text{Op}_\beta \Downarrow \ldots$$

where the indices (i.e., $\alpha$, $\beta$, etc) are drawn from $I$. Our model is abstract with respect to a given program $P$, which means that we do not explicitly define $P\text{Init}$ or any of the $P\text{Op}_i$ in this paper. Rather, we define a frame condition that is independent of any user program and that expresses what it means for operations to be legal in this base-language view.

To define BaseView, we require sets and functions for modeling objects, values, and object attributes. The set:

$$\text{OBJECTJD}^\perp$$

models object references. It contains a distinguished element, $\text{objNULL}$, that represents an uninitialized object reference. The set $\text{OBJECTJD}$ is the set of non-null object references.

$$\begin{align*}
\text{OBJECTJD} & : \text{OBJECTJD}^\perp \\
\text{OBJECTJD} & : \mathcal{P} \cup \text{OBJECTJD}^\perp \\
\text{OBJECTJD} & = \text{OBJECTJD}^\perp \setminus \{\text{objNULL}\}
\end{align*}$$

To model the meaning of an object attribute requires a space of legal values, which we define using the free type:

$$\text{VALUES} ::= \text{false} | \text{true} | \text{num} | \text{objref}$$

A value may be the boolean constants true (true) or false (false), a natural number (num), or an object reference (objref), which could be null. Because the null object reference appears often as a special case in our model, we give it the special name null, which is just an abbreviation for objref(objNULL).
We model BaseView as follows:

\[
\begin{align*}
\text{BaseView} & \quad \text{objClass : OBJECT_ID \rightarrow CLASS_ID} \\
& \quad \text{objData : OBJECT_ID \rightarrow ATTRIBUTE_ID \rightarrow VALUES} \\
\text{dom}(\text{objClass}) &= \text{dom}(\text{objData}) \\
\end{align*}
\]

Here, \textit{objClass} associates each object with the class that it instantiates, and \textit{objData} associates each object with a function that associates each attribute that the object contains with a value. The domains of \textit{objClass} and \textit{objData} are restricted to the non-null object references. This model captures the notion that a program is a dynamic collection of objects, each of which may be attributed. The elided invariant specifies that every object’s attributes conform to the class attributes associated with the object’s class and that the types of these attributes’ values conform to the attributes’ declared types.\footnote{This invariant uses functions defined in the basic type system to check the types of object attributes. The definition is straightforward, albeit tedious [48]. We omit the details for the sake of brevity, and because they are standard and are not germane to the language extension.}

A given user program engenders an initialization and a finite collection of operations, which can be formalized as Z operation schemas. Each such operation can update the \textit{objClass} and \textit{objData} functions, subject to the following condition: An operation may not change the class of an existing object. We specify this invariant as a frame condition, which must be conjoined with any operation in \(P\):

\[
\begin{align*}
\text{BaseViewFC} & \quad \Delta \text{BaseView} \\
\text{dom}(\text{objClass}) \cap \text{dom}(\text{objClass}') & \subseteq \text{dom}(\text{objClass} \cap \text{objClass}')
\end{align*}
\]

According to the Z convention for interpreting operation schema, unprimed and primed variables designate values before and after an operation, respectively (Appendix A). The invariant thus states that, if an object exists both before and after the operation (represented by an OBJECT_ID on which \textit{objClass} and \textit{objClass}' are both defined), then it instantiates the same class before the operation as after the operation. Frame conditions for the other views use invariants of a similar form.

\[
\begin{align*}
\text{ObjectAffectsRealm} & \equiv [ \Delta \text{BaseView}; \quad o : OBJECT_ID | \\
& \quad \text{dom}(\text{objData}(o) \setminus \text{objData}'(o)) \cap (\text{ConditionVars} \cup \text{UnitVars}) \neq \emptyset ] \\
\text{ObjectSleeps} & \equiv [ \Delta \text{BaseView}; \quad o : OBJECT_ID | \text{objData}'(o) = \text{objData}(o) ]
\end{align*}
\]

Table 3: Selected \textit{BaseView} operations

When we compose the views, we will need to define more complex frame conditions that either constrain or are predicated on a given base-level operation. Table 3 lists two operation schemas that are used as predicates during view composition. The first schema, \textit{ObjectAffectsRealm}, specifies the set of realm-affecting operations witnessed by
object \( o \), i.e., those operations that modify a condition variable or a unit variable of \( o \). Such an operation triggers the renegotiation of contracts, which may result in the reconfiguration of realms. The second schema, ObjectSleeps, specifies the set of operations in which no attributes of a given object \( (o) \) are modified. The object is said to sleep because it witnesses no activity.

### 5.2 Unit view

The unit view formalizes the interaction between objects and units without reference to any aspect of the base-language view. “Programs” in this view create and destroy objects and units subject to the following rule: Each unit contains a unique root object and any non-unit objects that are created by objects contained in the unit. Thus, a unit is a container that comprises a set of objects, with the property that an object in a unit can never become an element of another unit, and that contains a distinguished root object that is unique over the lifetime of the unit. To formalize these ideas, we introduce a set of identifiers to be used for denoting units:

\[
[UNIT,ID]
\]

The unit view relates each object to the unit that contains it and each unit to its root object.

\[
\begin{align*}
\text{UnitView} & \\
objUnit & : \text{OBJECT,ID} \rightarrow \text{UNIT,ID} \\
unitRoot & : \text{UNIT,ID} \leftrightarrow \text{OBJECT,ID} \\
\text{dom}(unitRoot) &= \text{ran}(objUnit) \\
unitRoot^- & \subseteq objUnit
\end{align*}
\]

The invariant specifies that every unit contains (at least) its root object. Moreover, because \( objUnit \) is a function, an object cannot be a member of two units simultaneously, and because \( unitRoot \) is injective, each unit has a unique root object.

To prevent objects in one unit from migrating into another, and to insure that the root object of a unit does not change over time, we use a frame condition:

\[
\begin{align*}
\Delta \text{UnitView} & \\
\text{dom}(objUnit) \cap \text{dom}(objUnit') & \subseteq \text{dom}(objUnit \cap objUnit') \\
\text{dom}(unitRoot) \cap \text{dom}(unitRoot') & \subseteq \text{dom}(unitRoot \cap unitRoot')
\end{align*}
\]

The first invariant insures that every object that exists both before and after the operation maps to the same unit after the operation as before. When we compose all of the views, this property is used to infer that the set of units partitions the object space of a base-language program and that this partition is stable modulo creation and deletion of objects. The second invariant insures that every unit that exists both before and after the operation maps to the same root object after as before. One consequence of this definition is that the lifetime of a unit is bounded by the lifetime of its root object. When we compose all of the views, this property can be used to infer that a new unit container is created when a base-language operation instantiates a synchronization class.
Table 4 lists two schemas that are used during view composition. The schema `UnitHasObject` is used to test if an object `o` is in a unit `u`. The more interesting schema `UnitSleeps` specifies the set of operations that leave the structure of a unit unmodified. In these operations the unit is said to sleep.

### 5.3 Realm view

The realm view groups units into realms, which may grow or shrink over time. “Programs” in this view create and destroy units and realms and reconfigure realms by migrating units among them. With one exception, every realm is associated with a process. The exception is the null realm, which contains any unit that is not part of the realm of some process. Examples of such units are the `EVENT MANAGER` and `CHANNEL` units during stages 0 and 3 in Figure 3. Previously, we described how every unit contains a unique root object. Likewise, every non-null realm contains a unique root unit, whose lifetime coincides with that of the realm.

To formalize these ideas, we define a new set of identifiers:

\[
\text{REALMJD}^⊥
\]

Similar to `OBJECTJD^⊥`, we identify within `REALMJD^⊥` a distinguished element, in this case `realmNULL`, and define a set `REALMJD` to be `REALMJD^⊥` minus this distinguished element.

\[
\begin{align*}
\text{realmNULL} & : \text{REALMJD}^⊥ \\
\text{REALMJD} & : \mathbb{P} \text{REALMJD}^⊥ \\
\text{REALMJD} & = \text{REALMJD}^⊥ \setminus \{\text{realmNULL}\}
\end{align*}
\]

We use the element `realmNULL` to define the null realm. Moreover, because each process has a unique realm, we use elements in the set `REALMJD` to identify processes.

We then define a schema that relates each unit to the realm that owns it and relates each non-null realm to its root unit.

\[
\begin{align*}
\text{RealmView} & \\
\text{unitRealm} & : \text{UNITJD} \rightarrow \text{REALMJD} \\
\text{realmRoot} & : \text{REALMJD} \rightarrow \text{UNITJD} \\
\text{dom}(\text{realmRoot}) & = \text{ran}(\text{unitRealm}) \setminus \{\text{realmNULL}\} \\
\text{realmRoot}^{-1} & \subseteq \text{unitRealm}
\end{align*}
\]

The first invariant guarantees that any unit not specifically associated with a process is in the null realm. In the sequel, we define realm programs that configure realms according to minimal-sufficiency and reduction criteria with
no mention of the null realm. The first invariant implicitly assigns units not designated by other processes to be in the null realm.

Analogous to a root object in the unit view, a root unit in the realm view is bound to the same realm throughout its lifetime.

\[
\Delta \text{RealmView} \subseteq \text{dom}(\text{realmRoot}) \cap \text{dom}(\text{realmRoot'}) \subseteq \text{dom}(\text{realmRoot} \cap \text{realmRoot'})
\]

Table 5 lists three schemas that are used in view composition. Observe that, by convention, schemas that refer to the full set of realms (REALMJD\(^{-}\)) are prefixed with the name “Realm”, whereas schemas that refer to the restricted set of realms (REALMJD) are prefixed with the name “Process”. The schema RealmHasUnit is used to test if a unit \(u\) is in a realm \(r\). The schema RealmReconfigures specifies the set of operations that modify the shape of a realm. Finally, schema ProcessInitializes specifies how to initialize the realm of a new process prior to any contract negotiation. Such a realm contains only its root unit.

\[
\begin{align*}
\text{RealmHasUnit} & \triangleq [\text{RealmView}; r : \text{REALMJD}^{-}; u : \text{UNITJD} | r = \text{unitRealm}(u)] \\
\text{RealmReconfigures} & \triangleq [\Delta \text{RealmView}; r : \text{REALMJD}^{-} | \text{unitRealm} \triangleright \{r\} \neq \text{unitRealm'} \triangleright \{r\}] \\
\text{ProcessInitializes} & \triangleq [\text{RealmView'}; p : \text{REALMJD} | (\{p\} \triangleleft \text{realmRoot'}) = \text{unitRealm'} \triangleright \{p\}]
\end{align*}
\]

Table 5: Selected RealmView operations and schemas.

5.4 Free program model of contract-aware programs

We model the behavior of a contract-aware program as the simultaneous execution of programs in each of the three views, subject to certain composition constraints. These constraints fall into two categories—those that relate the data in one view to data in another, and those that coordinate operations among the various views. Jackson advocates specifying constraints in the first category by conjoining the views and imposing so-called inter-view invariants and the second by joining operations [29].

The schema FreeProgramState composes the three views via inter-view invariants to yield a model of the state of a computation that incorporates units and realms such that the creation and deletion of units and realms is causally connected to the creation and deletion of objects in the base program.
Table 6: Table of selected free view operations and schemas.

<table>
<thead>
<tr>
<th>FreeProgramState</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseView</td>
</tr>
<tr>
<td>UnitView</td>
</tr>
<tr>
<td>RealmView</td>
</tr>
<tr>
<td>(\text{dom}(\text{objData}) = \text{dom}(\text{objUnit}))</td>
</tr>
<tr>
<td>(\text{dom}(\text{objClass} \triangleright \text{SynchClasses}) = \text{ran}(\text{unitRoot}))</td>
</tr>
<tr>
<td>(\text{dom}(\text{unitRealm}) = \text{dom}(\text{unitRoot}))</td>
</tr>
<tr>
<td>(\text{dom}(\text{objUnit} \triangleright \text{objClass} \triangleright \text{ProcessClasses}) = \text{ran}(\text{realmRoot}))</td>
</tr>
</tbody>
</table>

The first invariant requires that objects in the base-language view coincide with objects in the unit view. Consequently, units partition the set of base-language objects. The second invariant requires every base-language object that instantiates a synchronization class to be the root of some unit in the unit view. Consequently, when a base-language operation instantiates a synchronization class, a new unit is created with the object instance as the root; and, when a base-language operation deletes an instance of a synchronization class, the corresponding unit is deleted. The third invariant requires that units in the unit view coincide with units in the realm view. The fourth invariant requires every base-language object that instantiates a process class to coincide with a process in the realm view.

Table 6 lists several operation schemas over FreeProgramState. Note that operations in this space are formed by joining operations and conditions from the constituent views. Schema RealmSleeps specifies the set of operations for which no object or unit in a given realm is modified. Schema NullRealmSleeps specializes RealmSleeps to the case when \(r\) is the null realm, resulting in an invariant that must be true of all operations in a contract-aware program. By contrast, schema ProcessSleeps specializes RealmSleeps by limiting the domain of realms to only those that correspond to processes. In the sequel, we will use this schema to specify that disabled processes must block during an operation. Schema ProcessAffectsRealm specifies the operations in which a process is executing code in an object that performs a
realm-affecting operation. Schema ProcessRealmStable specifies the operations in which the realm of a process is the same both before and after the operation unless the operation comprises a BaseView operation that is realm-affecting. In the sequel, we use this schema to state that the realm of a running process is stable according to these conditions. Finally, schema ProcessInitializesReduced merely promotes the initialization schema for processes into an operation over FreeProgramState. Notice that the realm of an initialized process is trivially reduced because it contains only the root unit.

Following our previous conventions, we define constraints on operations using a frame condition, which when conjoined with a given BaseView operation will modify the appropriate units and realms.

\[
\text{FreeProgramStateFC} \equiv [ \Delta \text{FreeProgramState} \mid \text{BaseViewFC} \\
\quad \land \text{UnitViewFC} \\
\quad \land \text{RealmViewFC} \\
\quad \land \text{NullRealmSleeps} ]
\]

This condition subsumes the frame conditions defined for each individual view and adds the constraint that the null realm sleeps, which means that no operation may modify a unit or an object when that unit or object is in the null realm. Programs in FreeProgramState may create, destroy, and modify objects, units, and realms, subject to the afore-mentioned constraints, but this view cannot capture the reconfiguration of realms and the enabling/disabling of processes subject to the satisfaction of contracts. To complete the model of contract-aware programs requires reasoning about the status and satisfaction of contracts, which we describe in the next section.

6 Formal semantics of concurrency constraints

Recall that a process is enabled if its realm is minimally sufficient—that is, if its realm is sufficient and if removing any units from the realm would render the realm insufficient. Intuitively, a set avail of units is sufficient if it contains all the units needed to satisfy the concurrency constraints associated with each unit in avail. Recall also that the realm of a disabled process must be reduced—that is, be purged of every unit that is not required to satisfy a contract that is in force. To precisely define sufficiency and reduction requires formalizing what it means to satisfy a concurrency constraint. This formalization is accomplished by specifying: (1) an interpretation, which is a structure that is used to assign truth values to the atomic propositions of a constraint, and (2) a proof system, which shows how to demonstrate that a constraint is satisfied in a given interpretation. To motivate the structure of our interpretations, we appeal to a complex example involving two processes that negotiate for units in a linked structure.

6.1 Running example

Consider an application with two processes, with realms \( R_1 \) and \( R_2 \), that negotiate for units in a complex linked structure with non-trivial synchronization contracts. The time-series diagram in Figure 5 depicts the evolution of these realms, where \( u_0 = \text{realmRoot}(R_1) \), and \( u_5 = \text{realmRoot}(R_2) \). Initially, \( R_2 \) is minimally sufficient; however \( R_1 \) is not because unit \( u_1 \) has an unsatisfied contract for \( u_2 \), which is not in the realm. \( R_1 \) cannot acquire unit \( u_2 \) until
its contract can be satisfied, i.e., until \( R_2 \) releases unit \( u_4 \). Because it could not be made minimally sufficient, \( R_1 \) is reduced, which means it contains no superfluous units, such as \( u_2 \), only units that if removed would violate a contract that is in force. Notice in this case that \( u_1 \) is not superfluous because the root of \( R_1 \) (\( u_0 \)) has a contract that is in force.

Step one shows how the configuration of realm \( R_2 \) changes when \( u_5 \) sets its condition variable (\( b \)) to false. Realm \( R_2 \) contracts, migrating \( u_3 \) and \( u_4 \) into the null realm, after which \( R_2 \) is once again minimally sufficient. With the release of \( u_4 \), \( R_1 \) may now acquire \( u_2 \) and \( u_4 \) to become sufficient (Figure 5(c)). Step three illustrates the results of two realm-affecting operations: In \( R_1 \), \( u_2 \) sets the condition variable \( ok \) to false, and in \( R_2 \), \( u_5 \) sets its condition variable \( b \) to true. Notice that the change to \( u_2.ok \) does not invalidate the constraint of \( u_1 \) because conditions in conditional unit references can only defer migration into a realm; they do not cause units to be released. Thus, the change to \( u_2 \) has no effect on the contents or minimal sufficiency of \( R_1 \). The change to \( u_5 \) has no effect on the contents of \( R_2 \), but the status of \( R_2 \) changes from minimally sufficient to reduced.

Step 4 (Figure 5(e)) depicts a configuration of realms during the renegotiation process, in which both realms are reduced. Because \( u_1.a \) has been set to false, \( R_1 \) no longer needs \( u_2 \); thus it migrates both \( u_2 \) and \( u_4 \) to the null realm. Notice that \( u_1 \) cannot be released because it is needed to satisfy a contractual obligation with \( u_0 \), which is still in the realm. At this point, both \( R_1 \) and \( R_2 \) are competing for unit \( u_3 \). Step 5 depicts one of the two possible outcomes.

This example illustrates two points relative to constraint interpretation. First, constraint satisfaction is used to judge one of two different properties of a realm, sufficiency and reduction, which appeal to the satisfiability of constraints of the units in a realm in different ways: In a minimally sufficient realm, the constraint of every unit in realm set must be satisfied; whereas in a reduced realm, some units may have unsatisfied constraints. For example, in Figure 5(e), realm \( R_1 \) contains unit \( u_1 \), whose contract is not satisfied because \( u_3 \) is not in the realm. In both cases, the satisfiability of a constraint is judged in the context of a set of units, which we henceforth call the available set; however, what constitutes the available set will differ if one is judging sufficiency or reduction. When judging sufficiency, the available set is just the set of units in the candidate realm. For example, the sets \( R_1 \) in steps two and three and \( R_2 \) in steps zero, one, two, and five are all minimally sufficient realms, a fact that can be verified by checking the satisfaction of the contracts of each unit in these realms, using the realm as the available set. By contrast, when judging the reduction of a realm \( R' \) to \( R \), the available set is \( R \) unioned with the set of all units outside of \( R' \). For example the realm \( R_1 \) in step four is the reduction of the realm in \( R_1 \) in step three. To verify this fact, note that the contract of \( u_0 \) is satisfied because \( u_1 \) is in the realm at step four, and the contract of \( u_1 \) is satisfied because \( u_3 \) was not in \( R_1 \) at step three. Constructing the available set in this manner ensures that a reduction does not release a unit that is needed to satisfy a contract that was in force before the operation and that remains in force afterward.

Second, constraints involving conditional unit references can only be judged using knowledge about the contents of the realm prior the operation that causes the reassessment (and potential reconfiguration) of the realm. For example, consider the contract between units \( u_1 \) and \( u_2 \) in step 3 of Figure 5(d). The constraint,

\[
a \Rightarrow left \text{ when } left.ok
\]

is a parameterized conditional contract with parameter \( a \) and condition \( left.ok \). Because the parameter \( (u_1,a) \)
evaluates to true, the constraint should be true if \( u_2 \) is already in the realm that contains \( u_1 \) or if \( u_2 \) is available for migration and the condition \((u_2, \text{ok})\) is true. In this case, the condition is false, but \( u_2 \) was already in the realm; thus the constraint evaluates to true. Notice that to judge the constraint required consulting the contents of the realm prior to the operation. Thus, our interpretation structure includes a set, called \( \text{owns} \), that retains the contents of the old realm and whose units will satisfy a conditional unit reference even if the condition evaluates to false.

### 6.2 Interpretations

An interpretation is a structure that is used to assign truth values to the atomic propositions of conditions and constraints. Interpretations are defined as follows:

<table>
<thead>
<tr>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit : UNIT_ID</td>
</tr>
<tr>
<td>avail : ( \mathbb{P} ) UNIT_ID</td>
</tr>
<tr>
<td>owns : ( \mathbb{P} ) UNIT_ID</td>
</tr>
<tr>
<td>root : UNIT_ID ( \mapsto ) OBJECT_ID</td>
</tr>
<tr>
<td>data : OBJECT_ID ( \mapsto ) ATTRIBUTE_ID ( \mapsto ) VALUES</td>
</tr>
</tbody>
</table>

Here, \( \text{unit} \) is the unit that is subject to the constraint being verified, \( \text{avail} \) is the available set, \( \text{owns} \) is the set of units that are known to be in the realm prior to the operation that is causing the reassessment of the realm, and \( \text{root} \) and \( \text{data} \) are functions for extracting the root object of a unit and for looking up the values of object attributes respectively. The \( \text{root} \) and \( \text{data} \) functions are analogous to the \text{unitRoot} and \text{objData} functions in the unit and base views. The invariants require \( \text{avail} \) to be drawn from the space of active units and require \( \text{unit} \) to be available.

As mentioned previously, the contents of \( \text{avail} \) will differ depending on whether the interpretation is being used to verify the sufficiency or reduction of a realm. For example, when checking if a set \( R \) is a sufficient realm, a unit is considered available if it is an element of \( R \). By contrast, when checking if \( R \) is a core reduction of \( R' \), a unit is considered available if it is in \( R \) or in the set of units outside of the uncontracted realm (i.e., anything except \( R' \setminus R \)).

Consider verifying the constraint of unit \( u_0 \) to assess the sufficiency of \( R \) in the initial configuration (Figure 5(a)). For this constraint to be satisfied, the unit whose root is the object referenced by \( y \) must be in \( R_1 \). This verification task requires looking up the value of \( y \) in the root object of \( u_0 \) and checking that either the value is \text{null} or that it references an object that is the root of some available unit, in this case unit \( u_1 \). The interpretation that would be constructed to verify this constraint is:

\[
(\text{unit} \equiv u_0, \text{avail} \equiv \{u_0, u_1\}, \text{owns} \equiv \{u_0, u_1\}, \text{root} \equiv \text{unitRoot}, \text{data} \equiv \text{objData})
\]

where \( \text{unitRoot} \) and \( \text{objData} \) are the functions defined in \text{UnitView} and \text{BaseView} respectively. In this example, \( y \) maps to the root object of unit \( u_1 \), which is in the set \( \text{avail} \) under this interpretation. Thus, this interpretation verifies \( u_0 \)'s constraint.
The *owns* set is required to assign a truth value to conditional unit references. Recall from Section ?? that a conditional unit reference behaves just like a regular unit reference, except that migration of the referent into a realm is deferred until the condition is satisfied. Once the referent is migrated, the conditional unit reference remains true until the referent is migrated from the realm, regardless of the condition’s value. Suppose, for example, we want to check the constraint of unit $u_1$ at step 2 in Figure 5(c). The interpretation that would be used is:

\[
\langle \text{unit} \Rightarrow u_1, \text{avail} \Rightarrow \{u_0, u_1, u_2, u_4\}, \text{owns} \Rightarrow \{u_0, u_1\}, \text{root} \Rightarrow \text{unitRoot}, \text{data} \Rightarrow \text{objData} \rangle
\]

Notice that *owns* records the contents of the $R_1$ prior to the release of $u_4$. Because $u_2$ is in *avail* but not in *owns*, the condition $u_2.ok$ must be checked when verifying $u_1$’s constraint.

### 6.3 Satisfaction relation

The satisfaction relation ($\models$) relates interpretations to the constraints that they satisfy. We introduce the satisfaction relation using an axiomatic description:

\[
\mid \text{Interpretation} \models \text{Constraint}
\]

and we define it using a natural deduction system similar to that used in defining typing judgements. The binding:

\[
\langle \text{unit} \Rightarrow \text{unit}, \text{avail} \Rightarrow \text{avail}, \text{owns} \Rightarrow \text{owns}, \text{root} \Rightarrow \text{root}, \text{data} \Rightarrow \text{data} \rangle
\]

declares an instance of schema *Interpretation*, whose *unit*, *avail*, *owns*, *root*, and *data* fields are bound to the values *unit*, *avail*, *owns*, *root*, and *data* respectively. Suppose $C$ is the concurrency constraint associated with $u$. Then we write:

\[
\langle \text{unit} \Rightarrow \text{unit}, \text{avail} \Rightarrow \text{avail}, \text{owns} \Rightarrow \text{owns}, \text{root} \Rightarrow \text{root}, \text{data} \Rightarrow \text{data} \rangle \models C
\]

to mean that this interpretation satisfies $C$.

As with typing judgements, we formally define the satisfaction relation by induction on the structure of $C$. Table 7 lists the rules that govern atomic conditions and Table 8 lists the rules that govern atomic constraints. The inference rules that govern the boolean connectives are standard [48]. For brevity in the rules, we abbreviate the binding syntax, indicating only the values. A constraint $C$ is valid if it evaluates to true under the following substitution:

1. Each local condition variable $x$ in $C$ is replaced with the value the root object of $u$ assigns to $x$ (Table 7, [lcondM]).

2. Each remote condition variable $x,y$ in $C$ is replaced with:
   - **true** if the root of $u$ assigns `null` to $x$ (Table 7, [rcondM1]);
   - **true** if the root of $u$ assigns to $x$ a reference to a unit $u' \in \text{avail}$ whose root object assigns `true` to $y$ (Table 7, [rcondM1]); and
   - **false** otherwise
3. Each simple unit reference \( x \) in \( C \) is replaced with:

- \( \text{true} \) if the root of \( u \) assigns \( \text{null} \) to \( x \) (**Table 8**, [uvConst1]);
- \( \text{true} \) if the root of \( u \) assigns to \( x \) a reference to a unit \( u' \in \text{avail} \) (**Table 8**, [uvConst2]); and
- \( \text{false} \) otherwise

4. Each conditional unit reference \( x \text{ when } c \) in \( C \) is replaced with:

- \( \text{true} \) if the root \( u \) assigns \( \text{null} \) to \( x \) (**Table 8**, [whConst1]);
- \( \text{true} \) if the root of \( u \) assigns to \( x \) a reference to some unit \( u' \in \text{avail}, u' \not\in \text{owns} \), and the referent condition \( (c) \) is satisfied (**Table 8**, [whConst2]);
- \( \text{true} \) if the root of \( u \) assigns to \( x \) a reference to some unit \( u' \in \text{avail} \) and \( u' \in \text{owns} \) (**Table 8**, [whConst3]); and
- \( \text{false} \) otherwise

### 6.4 Realm predicates and contract negotiation

As a contract-aware program executes, contracts are dynamically renegotiated, which means units dynamically migrate into and out of the realms of processes. This migration is governed by two major requirements. First, the realm of an enabled process should be the minimal set of units such that the contract associated with each unit in the set is satisfied. Second, a unit should never migrate out of a realm if it is required to satisfy a contract that is already in force. Both of these requirements are necessary to ensure safety. In the first case, if a process executes in a realm that is not minimal, the process could access a unit to which it has no access rights—access rights being granted explicitly through the negotiation of contracts. In the second case, if a unit that was acquired to satisfy a contract is allowed to migrate out of its owner’s realm, then another process could access the unit and thus violate the exclusion guaranteed by the contract. Both requirements refer to the *status* of contracts—e.g., the contract has been negotiated and is in force, the contract can be negotiated, etc. We formalize the status of contracts in terms of the satisfaction of constraints and codify the safety requirements as *realm predicates*.
A realm predicate is true if the constraint of each unit in the realm is satisfied under an appropriately constructed interpretation. We specify this predicate as a schema called \textit{RealmStatusPredicate} (Figure 6). The invariant instantiates an \textit{Interpretation} for each unit belonging to \textit{realm} and appeals to the satisfaction relation to check that the interpretation satisfies that unit’s constraint. The expression $\theta$\textit{Interpretation} constructs the characteristic binding of schema \textit{Interpretation}. Characteristic bindings are schema instances that bind their variables to the values contained in variables of the same name and type that are in scope when the expression is evaluated [47]. In this context, $\theta$\textit{Interpretation} constructs the binding:

\begin{equation}
\{ \text{unit} \Rightarrow \text{unit}, \text{avail} \Rightarrow \text{avail}, \text{owns} \Rightarrow \text{owns}, \text{root} \Rightarrow \text{root}, \text{data} \Rightarrow \text{data} \}
\end{equation}

where the values are those of the bound variable \textit{unit} and of the \textit{RealmStatusPredicate} variables \textit{avail}, \textit{owns}, \textit{root}, and \textit{data}. We named the variables in \textit{RealmStatusPredicate} to coincide with the names of the variables in \textit{Interpretation} in order to use the $\theta$ operator to instantiate interpretations.

We specialize \textit{RealmStatusPredicate} into useful predicates by constraining the contents of the three sets—\textit{realm}, \textit{avail}, and \textit{owns}. For example, schema \textit{StatusSufficient} specializes \textit{RealmStatusPredicate} by equating the \textit{avail} and \textit{realm} sets. Thus, \textit{StatusSufficient} will evaluate to true if the constraint of each element in \textit{realm} is satisfied by an interpretation that treats only the elements in \textit{realm} as available. As the name suggests, this schema precisely defines the conditions under which a realm is sufficient.

By contrast, schema \textit{StatusContracted} specializes \textit{RealmStatusPredicate} to capture what is true of a realm that has released units without violating contracts that are currently in force. The first invariant states that \textit{realm} must be a subset of \textit{owns}, i.e., that the realm can only shrink. The more complex invariant fixes the value of the \textit{avail} set to include every unit in \textit{realm} and every unit that is not released in contracting \textit{owns} to \textit{realm}. This construction prevents the release of units that must remain in the realm to satisfy the requirements of a contract that was previously

\begin{table}[h]
\centering
\begin{tabular}{ll}
\hline
\textit{data} (\textit{root} (\textit{unit})) (\textit{x}) = \texttt{null} & $\exists u' : \texttt{avail} \cap \texttt{dom} (\textit{root})$ \\
\hline
\{ \textit{unit}, \textit{avail}, \textit{owns}, \textit{root}, \textit{data} \} $\models \text{unitvar} (\textit{x})$ & $\textit{data} (\textit{root} (\textit{unit})) (\textit{x}) = \texttt{objref} (\textit{root} (u'))$ \\
\hline
\hline
\textit{data} (\textit{root} (\textit{unit})) (\textit{x}) = \texttt{null} & $\neg \exists u' : \texttt{owns}$ \\
\hline
\{ \textit{unit}, \textit{avail}, \textit{owns}, \textit{root}, \textit{data} \} $\models \text{when} (\textit{x}, \textit{c})$ & $\textit{data} (\textit{root} (\textit{unit})) (\textit{x}) = \texttt{objref} (\textit{root} (\textit{u'}))$ \\
\hline
\hline
\end{tabular}
\caption{Semantics of local satisfaction for atomic constraints}
\end{table}
To see why this construction results in the desired property, consider the configuration in Figure 5, which depicts a realm in need of contraction because of a change in the parameters to the contract of unit $u_1$. This change causes the renegotiation of every contract in this realm except for $u_0$’s contract for $u_1$, which remains in force. Consequently, the contracted realm should be the set $\{u_0, u_1\}$. To verify this set is a contracted realm, we set $realm$ to $\{u_0, u_1\}$ and set $owns$ to the contents of the uncontracted realm, i.e., $\{u_0, u_1, u_2, u_4\}$. Then $avail = \{u_0, u_1, u_3, u_5\}$. Notice that the interpretation whose $unit$ value is $u_0$ satisfies $u_0$’s constraint because $u_1 \in avail$. Moreover, the interpretation whose $unit$ value is $u_1$ satisfies $u_1$’s constraint because $u_3 \in avail$. By including in $avail$ those units that are outside of the uncontracted realm, the resulting interpretation will satisfy the constraint of a unit whose constraint could not have been satisfied by an interpretation constructed using the $StatusSufficient$ predicate. This is necessary to retain units that are needed to satisfy a contract that is in force but which may have contracts that cannot be satisfied.

Notice that $StatusContracted$ would be false if the value for $realm$ contained only the singleton set $\{u_0\}$. Notice also that $StatusContracted$ returns true for any set that contains $\{u_0, u_1\}$, even when $realm = owns$. Finally, notice that some seemingly silly sets, e.g., $\{u_4\}$ would also cause $StatusContracted$ to return true. To prevent these anomalies, we ensure that every $realm$ contains at least the root unit of the process whose realm we wish to contract.

A unit set is minimally sufficient if it is sufficient and if no proper subset containing the root unit of the process is also sufficient. Schema $StatusMinimallySufficient$ formalizes this notion. The last line of the invariant asserts that the predicate obtained by replacing all occurrences of the variable $realm$ in $StatusSufficient$ with the variable $smaller$ is true. Likewise, schema $StatusReduced$ asserts that the predicate obtained by replacing all occurrences of the variable $realm$ in $StatusContracted$ with the variable $smaller$ is true.

Finally, the schema $ProcessRealmStatus$ is used to promote a $RealmStatusPredicate$ into a schema that specifies the set of $FreeProgramState$ operations that complete with the $RealmStatusPredicate$ true. The schema follows the profile of a promotion schema, including the larger state space ($\Delta FreeProgramState$), $RealmStatusPredicate$, and a process identifier ($p$) to which the predicate is to apply. The invariant sets $realm$ to be the set of units in the realm of $p$ after the operation, $owns$ to be the set of units in the realm of $p$ before the operation, and $root$, $data$, and $class$ to be the functions from the appropriate views. Table 9 defines two schemas, $ProcessSufficesRealm$ and $ProcessReducesRealm$, which promote the realm predicates $StatusMinimallySufficient$ and $StatusReduced$ into $FreeProgramState$ operations. These schemas are used to define the second-tier model, which extends programs in $FreeProgramState$ with contract negotiation and process enabling/disabling.
\[
\begin{align*}
\text{EnabledProcessExecutesInStableRealm} & \equiv \left[ \Delta \text{SynchronizedProgramState}; p : \text{REALM JD} \mid p \in \text{EnabledProcs} \Rightarrow \text{ProcessRealmStable} \right] \\
\text{DisabledProcessDoesNotExecute} & \equiv \left[ \Delta \text{SynchronizedProgramState}; p : \text{REALM JD} \mid p \in \text{DisabledProcs} \Rightarrow (p \in \text{Procs}' \Rightarrow \text{ProcessSleeps}) \right] \\
\text{SufficientRealmEnablesItsOwner} & \equiv \left[ \Delta \text{SynchronizedProgramState}; p : \text{REALM JD} \mid \text{ProcessSufficesRealm} \Leftrightarrow p \in \text{EnabledProcs}' \right] \\
\text{DisablingAProcessReducesItsRealm} & \equiv \left[ \Delta \text{SynchronizedProgramState}; p : \text{REALM JD} \mid p \in \text{DisabledProcs}' \Rightarrow \\
& \quad (p \notin \text{Procs} \land \text{ProcessInitializesReduced} \\
& \quad \lor \\
& \quad p \in \text{Procs} \land \text{ProcessReducesRealm}) \right]
\end{align*}
\]

Table 10: Table of key frame conditions for reasoning about the effects of contract-aware programs.

7 Complete model of contract-aware programs

We now complete our model of the behavior of contract-aware programs by adding the second tier, which shows how contract negotiation enables and constrains \textit{FreeProgramState} operations. The state space of a contract-aware program is characterized by the schema:

\[
\begin{align*}
\text{SynchronizedProgramState} & \equiv \text{FreeProgramState} \\
\text{FreeProgramState} & \\
\text{Procs} : \mathbb{P} \text{ REALM JD} \\
\text{EnabledProcs} : \mathbb{P} \text{ REALM JD} \\
\text{DisabledProcs} : \mathbb{P} \text{ REALM JD} \\
\text{EnabledProcs} & \subseteq \text{Procs} \\
\text{DisabledProcs} & = \text{Procs} \setminus \text{EnabledProcs} \\
\text{Procs} & = \text{dom} \left( \text{realmRoot} \right)
\end{align*}
\]

which extends \textit{FreeProgramState} with a set \textit{Procs} of active processes, which is partitioned into a subset of processes that are enabled and a subset that are disabled. Active processes are processes that have been created and have not yet terminated. The first two invariants partition the set of active processes into \textit{EnabledProcs} and \textit{DisabledProcs}, and the third invariant requires processes in the free program to coincide with active processes in the scheduler view. Table 10 lists four schemas that enable and constrain \textit{FreeProgramState} operations based on the scheduling status (i.e., enabled or disabled) of a process. The first two constrain the activity of a process based on its status; whereas the second two specify how to update the realm and status of a process after each operation.

Our contract model dictates that the realm of an enabled process will be stable modulo the migration that occurs in response to a realm-affecting operation in the base program. Our semantics ensure that only a realm-affecting operation may cause the reconfiguration of a minimally sufficient realm. Upon executing such an operation, the run-time system tries to assemble a minimal and sufficient realm for the process or for some other (waiting) process by migrating units
among their realms. We formalize this behavior using the frame condition \textit{EnabledProcess Executes In Stable Realm} (Table 10). When an enabled process executes an operation that is not realm-affecting, no units will migrate into or out of its realm.

A disabled process may do only one of two things, sleep or terminate. A sleeping process cannot modify any of the objects or units in its realm; however, it may reconfigure its realm to incorporate units that are released by other processes. This must be the case in order for units to migrate into the realm of a blocked process so as to complete its realm and thus enable it. The operation schema \textit{DisabledProcess Does Not Execute} specifies that a disabled process either terminates (in which case \( p \notin \text{Procs} \)) or sleeps (Figure 10).

An operation should leave a process enabled if and only if the realm of the process is minimally sufficient. The frame condition \textit{Sufficient Realm Enables Its Owner} codifies this constraint (Table 10). Because this frame condition involves an if and only if (i.e., \( \leftrightarrow \)), any process whose realm cannot be made minimally sufficient will be disabled. Our contract model requires that such a process reduce its realm, which we specify using the frame condition \textit{Disabling A Process Reduces Its Realm}. In English, the predicate on the right hand side of the implication states that if process \( p \) is created by this operation (i.e., \( p \notin \text{Procs} \)) then its realm will be initialized to contain only its root unit. On the other hand, if \( p \) persists through this operation (i.e., \( p \in \text{Procs} \)) then the process must reduce its realm. The following frame condition specifies that an operation over \textit{Synchronized Program State} must impose the frame conditions outlined above on each active process:

\[
\text{Synchronized Program State FC} \equiv \left[ \Delta \text{Synchronized Program State} \mid \right.
\begin{align*}
\text{Free Program State FC} \land \\
(\forall p : \text{Procs} \cup \text{Procs}') \bullet \\
\text{Enabled Process Executes In Stable Realm} \land \\
\text{Disabled Process Does Not Execute} \land \\
\text{Sufficient Realm Enables Its Owner} \land \\
\text{Disabling A Process Reduces Its Realm} \right]
\]

Finally, the operations of a base-language program are promoted into operations over \textit{Synchronized Program State} by conjoining them with the frame conditions of each view and the frame condition \textit{Synchronized Program State FC}. This collection of frame conditions thus cleanly defines the effect on concurrency and synchronization of executing base-language operations in a language-neutral manner. Before concluding, we note that the specification does not prescribe how non-root objects join a unit. We saw an example of how this might occur with the \texttt{Display} object in Figure 2. Our informal convention has been that, when an object of a non-synchronization class is created, the object becomes a part of the unit that created it. However, the formal model does not prescribe this mechanism because we envision that other mechanisms may be useful; for instance, a declaration might designate a unit in which to place a non-unit object.

8 Discussion

Our model extends traditional object-oriented languages with synchronization contracts. Specifically, it introduces declarative concurrency constraints to describe conditions under which units depend on one another. The run-time
system uses concurrency constraints to automatically determine when to migrate units in and out of realms and how to schedule processes to prevent race conditions and avoid many classes of deadlock and starvation. This paper has presented a language-neutral formal definition of the semantics of this model. The definition is structured using multiple views that compose by conjunction to facilitate understanding and permit reuse.

In the remainder of this section, we explain why we chose not to model inheritance (Section 8.1). We then compare our model-theoretic semantics of concurrency with other semantic models (Section 8.2) and other approaches to separating concurrency concerns from core functionality (Section 8.3). Finally, we conclude with a discussion of open research questions (Section 8.4).

8.1 Inheritance

We do not model class inheritance in the base-language view of our formal semantics in order that the semantics will be abstract with respect to the base language. Languages differ in how they implement inheritance. We therefore assume only that a programmer associates a concurrency constraint with each synchronization class. However, because concurrency constraints are attached to classes, and not regions of code, the constraints integrate very naturally with the inheritance mechanisms of most languages. For example, in our extension to Eiffel, a derived class inherits the concurrency constraints of its ancestors and may specialize these inherited constraints by adding a further constraint of its own. Inherited constraints compose by conjunction. That is, the constraint of a derived class is the conjunction of all inherited constraints and any constraint that it adds.

Inheritance provides a powerful abstraction mechanism, facilitating extension and reuse of software. However, new methods or data in a derived class may also require redefinition of methods in a base class. The generalized body anomaly [25] is a common problem that arises when synchronization code is placed in the body of a base-class method and a derived class must change this synchronization behavior—say, to synchronize on an additional object. In such cases, the derived class must redefine the method, copying and suitably modifying the code from the base class. Our model of synchronization contracts is not immune to the generalized body anomaly. However, we consider good programming practice dictates that condition variables, procedures that modify them, and the associated concurrency constraints be encapsulated within a synchronization class that clients can inherit—that is, within a base class. This practice isolates, in a single class, the code that might need to be overridden to adjust synchronization behavior. It also reduces the need to replicate synchronization contracts in multiple clients because the clients can all inherit from the same base class. The main practical drawback of this method is that it requires multiple implementation inheritance, which is not supported by all object-oriented languages.

8.2 Semantics of concurrency as a language extension

As discussed in the introduction, an object-oriented programming language that has been extended with synchronization contracts is a muti-paradigm programming language; it provides language features for expressing core functionality in one programming paradigm and language features for expressing synchronization in another. A multi-paradigm
programming language simplifies programming to the extent that it supports both separation of concerns, allowing the programmer to understand the code for individual concerns in isolation, and composition of concerns, allowing the programmer to understand how the code for individual concerns joins to form a program. Zave and Jackson therefore argue that “sub-languages” of a multi-paradigm language should denote model-based partial specifications, such that the conjunction of these partial specifications defines the behavior of the whole system [54, 55]. For this reason, we structured our semantics into separate views [29], each of which captures a different concern (e.g., functional code, synchronization units, realms, the run-time scheduler, etc.) and all of which compose by conjunction. Moreover, we define a minimal base-language view, which generalizes most object-oriented programming languages. Thus, by this technique, we have demonstrated that our notion of synchronization contracts integrates seamlessly with the features of a large class of object-oriented languages.

Others have used different techniques for defining the semantics of concurrency as a language extension. In an appendix of Concurrent Programming in ML, Reppy extends the formal semantics for a subset of Standard ML (SML) with combinators that express the semantics of threads and higher-order communication and synchronization primitives [44]. However, the combinators are designed for extending the operational definition of SML [37], and are not abstract with respect to a base language. In contrast, Noble, Holmes, and Potter present an algebraic model for concisely specifying and reasoning about equivalences among so-called exclusion constraints over method invocations in an aggregation hierarchy of objects [41]. Like our model, their algebra can be applied to programs written in many different object-oriented languages. Our work differs in how we abstract the base language—as a collection of objects and object links, instead of a collection of strict containment hierarchies and method names. Moreover, their exclusion constraints have not been used to extend a language; rather they are used as a design tool to reason about the optimal placement of lock statements in a containment hierarchy.

We layer views that formalize the semantics of synchronization contracts on top of a formal view of object-oriented languages. Abadi and Cardelli present a seminal theory of objects to formalize the semantics of object-oriented languages [1]. Their object calculi represent objects directly and provide operators for expressing central concepts of object orientation, such as inheritance, subtyping, polymorphism, and so forth. In contrast, our base-language view abstracts away much of this complexity on the grounds that these distinctions are not germane to the synchronization of processes under a contract model.

While our model-theoretic semantics is useful for understanding how concurrency constraints are integrated into an object-oriented language, the semantics does not enable execution, nor does it directly support concurrency analysis. These tasks require either an operational semantics of programs in our model or a denotational semantics that translates such programs into a more suitable concurrency-modeling notation. Traditionally, notations such as CSP [24] and those based on process algebra (e.g., CCS [36], LOTOS [10], or π-calculus [38]) have been used to model concurrency. However, because a unit encapsulates both data and behavior, and because a process may need to acquire multiple units atomically, unit negotiation is difficult to model via the synchronization of actions. We are currently investigating how to analyze programs in our model for general temporal properties.
8.3 Other approaches to separating synchronization and core computation

Others have investigated separating synchronization concerns from a program’s core functionality. Lopes [32] developed a language for programming so-called coordinator classes, which define reusable supplier-side synchronization logic as aspects that can be woven into the implementation of arbitrary classes. While this approach separates synchronization logic from the core functionality, the logic is still expressed in terms of low-level synchronization primitives. Thus, this is not a multi-paradigm approach. We are aware of two multi-paradigm approaches that separate client-side synchronization logic using explicit specifications in lieu of code. Deng et al. specify the entry/exit logic for critical regions using a small collection of composable synchronization patterns [18]. This specification is used to automatically generate an explicit synchronization component, whose services are invoked when the client code enters and leaves a critical region. Moreover, the specification can be used in program analysis to create a model of the application that is suitable for analysis of concurrency properties, including liveness properties. Betin-Can and Bultan [7] specify so-called concurrency controllers using a notation that separates the interface and implementation of these controllers, thereby allowing a programmer to verify thread correctness using only these interfaces.

The synchronization specifications in these other approaches are not associated with individual synchronization units. Rather, they describe the behavior of an explicit component, which has its own synchronization contract in the form of an invocation interface that clients must respect. For example, in the concurrency-controllers approach, to verify thread correctness, a designer effectively verifies that client code ensures the requirements specified in the synchronization specification. Unlike our model, neither of these approaches guarantees freedom from data races, which may still occur if the designer improperly specifies an invariant or a concurrency controller, or fails to identify a critical region.

Finally, implementation techniques from database transactions have been used to implement conditional critical regions directly in existing programming languages [43, 22]. Under these approaches, a programmer identifies a critical region, which is then treated as a transaction which must run atomically with respect to other regions. These transactions are then implemented using non-blocking algorithms which compute the transaction in a local memory and commit by copying the results back to main memory if another thread has not entered the region in the interim. The most significant drawback to this approach is the limitation on operations that can appear in a critical region. Because a transaction may need to roll back and restart several times before it finally commits, the code within a critical region cannot perform I/O or create new threads. Also, there is no safeguard against a programmer failing to identify a critical region.

8.4 Future Research Directions

We are currently working to extend this research in several directions. First, we want to use these semantics to enable the static analysis of safety properties, specifically freedom from deadlock. Because our semantics is model theoretic, it should be easy to encode in a language such as Alloy, which would then enable the use of the Alloy analyzer [30] for a very coarse form of anomaly detection. For example, we might be able to demonstrate that a configuration of
units with cyclic dependencies cannot deadlock because the concurrency constraints are mutually exclusive. A more interesting analysis concerns the discovery of execution scenarios that lead to deadlock. By modeling the behavior of individual units as a labeled transition system, where states are labeled by configurations of condition variables and transitions represent condition-variable assignments, we could use tools such as SMV [34] or Spin [26] for deadlock analysis and counter-example detection.

Second, we are exploring how to further separate synchronization and core-computation concerns. Our current implementations require a designer to write code to explicitly represent and maintain so-called typestates over the course of a running computation. Typestates are properties of an object that capture information about its state while retaining the simplicity and feel of types [49, 17]. For example, in a system in which producer and consumer threads synchronize using a shared buffer, the typestate of the buffer might comprise one of three values: empty, full, or neither empty nor full. Our current implementations encode typestates using condition variables and maintain typestate consistency using explicit assignments to these condition variables. We are investigating how to integrate typestates into the static type system and how to specify typestate-maintenance logic as an abstract state machine whose nodes represent distinct typestate values. This specification will require some facility for associating typestates with the values of variables in the core computation and with regions of program code. An interesting research question will be whether these declarative facilities can completely obviate the need to interleave core-functional code with typestate-maintenance code.

Third, we are exploring how to integrate our model into object-oriented modeling notations (e.g., UML [11]). One possibility is to use UML instance diagrams to specify static configurations of units, using stereotypes to designate that an instance is a unit and OCL adornments to associate concurrency constraints with these units. We may also be able to use UML state diagrams to graphically depict the labeled transition systems in terms of the typestates that parameterize the synchronization behavior of units. Whereas state diagrams have been used to reason about concurrency and synchronization, most approaches assume that each state diagram describes an active object with message passing or synchronous events. By contrast, our approach may enable the use of state diagrams for reasoning about the behavior of concurrent programs in the shared-storage model.

Finally, we are also looking at how to extend our model to express higher-level contracts that negotiate among clients and suppliers in order to satisfy quality of service (QoS) requirements.

9 Acknowledgements

This work is supported in part by the National Science Foundation Grants EIA-0000433, CCR-9984726, and EIA-0130724; by the Defense Advanced Research Projects Agency; and by the Office of Naval Research Grant N00014-01-1-0744. Any opinions, findings, and conclusions or recommendations explained in this article are the authors’ and do not reflect the views of the supporting agencies.
References


A Glossary of Z notation used in the paper

We adopt the following notational conventions below: $X$ and $Y$ denote sets; $r$ and $s$ denote relations

Other Z notational conventions:

- for all variables in $decl$ satisfying $pred_1$, it is the case that $pred_2$ holds: $\forall decl \mid pred_1 \bullet pred_2$
- there exist variables in $decl$ satisfying $pred_1$ such that $pred_2$ holds: $\exists decl \mid pred_1 \bullet pred_2$
- the set of values $exp$ ranging over variables in $decl$ that satisfy $pred$: $\{decl \mid pred \bullet exp\}$
- given set, denoted by $X$

$$[X]$$
- schema named $Name$, binding $x$ to a value in $X$ and $y$ to a value in $Y$ subject to invariant $inv$:

```
Name
  x : X
  y : Y
inv
```
x : X     a variable declaration, x belongs to set X
P X     set of subsets of X
X \ Y    set subtraction
X \times Y    cross-product of X and Y
X \leftrightarrow Y    set of all relations between X and Y
X \rightarrow Y    set of partial functions from X to Y
X \rightarrowrightarrow Y    set of injective partial functions from X to Y
dom(r)    domain of r
tan(r)    range of r
r^\sim    relational inverse of r
r \circ s    relational composition of r and s
r \rhd Y    range restriction of r, tan(r \rhd Y) = Y
X \rhd r    domain restriction of r, dom(X \rhd r) = X

– conjunction of schemas Name1 and Name2, includes the declarations and invariants of both schemas and also the invariant inv:

\[
\begin{array}{c}
\text{Name} \\
\text{Name1} \\
\text{Name2} \\
\text{inv}
\end{array}
\]

– operation schema, contains two “copies” of schema Name, one of which has all its variables decorated with a prime—plain variables designate values before the operation and decorated variables designate values after the operation:

\[
\begin{array}{c}
\text{NameOp} \\
\text{\Delta Name}
\end{array}
\]
Figure 5: Running example of realm completion and contraction
<table>
<thead>
<tr>
<th>RealmStatusPredicate</th>
<th>ProcessRealmStatus</th>
</tr>
</thead>
<tbody>
<tr>
<td>realm : ( \mathcal{P} \text{UNIT_ID} )</td>
<td>(\Delta\text{FreeProgramState} )</td>
</tr>
<tr>
<td>avail : ( \mathcal{P} \text{UNIT_ID} )</td>
<td>RealmStatusPredicate</td>
</tr>
<tr>
<td>owns : ( \mathcal{P} \text{UNIT_ID} )</td>
<td>( p : \text{REALM_ID} )</td>
</tr>
<tr>
<td>root : UNIT_ID (\sim) OBJECT_ID</td>
<td>( \text{realm} = \dom(\text{unitRealm} \cup { p }) )</td>
</tr>
<tr>
<td>data : OBJECT_ID (\rightarrow) ATTRIBUTE_ID (\rightarrow) VALUES</td>
<td>(\text{owns} = \dom(\text{unitRealm} \cup { p }) )</td>
</tr>
<tr>
<td>class : OBJECT_ID (\rightarrow) CLASS_ID</td>
<td>( \text{root} = \text{unitRoot}' )</td>
</tr>
<tr>
<td>realm (\subseteq) (\dom(\text{root}))</td>
<td>(\text{data} = \text{objData}' )</td>
</tr>
<tr>
<td>(\forall) unit : realm (\bullet)</td>
<td>(\text{class} = \text{objClass}' )</td>
</tr>
<tr>
<td>(\theta) Interpretation (\models) (\text{constraint}(\text{class}(\text{root}(\text{unit}))))</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StatusSufficient</th>
<th>StatusMinimallySufficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealmStatusPredicate</td>
<td>StatusSufficient</td>
</tr>
<tr>
<td>avail = realm</td>
<td>(\neg (\exists) smaller : (\mathcal{P}) UNIT_ID</td>
</tr>
<tr>
<td></td>
<td>(\text{smaller} \subseteq \text{realm} \bullet)</td>
</tr>
<tr>
<td></td>
<td>(\text{StatusSufficient}[\text{smaller} / \text{realm}])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StatusContracted</th>
<th>StatusReduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealmStatusPredicate</td>
<td>StatusContracted</td>
</tr>
<tr>
<td>realm (\subseteq) owns</td>
<td>(\neg (\exists) smaller : (\mathcal{P}) UNIT_ID</td>
</tr>
<tr>
<td>avail = (\text{dom})(root) (\setminus) owns (\cup) realm</td>
<td>(\text{smaller} \subseteq \text{realm} \bullet)</td>
</tr>
<tr>
<td></td>
<td>(\text{StatusContracted}[\text{smaller} / \text{realm}])</td>
</tr>
</tbody>
</table>

Figure 6: Table of predicates for judging sufficiency and minimality of realms.