

Safe and Reliable Use of Concurrency in Multi-Threaded Shared-Memory Systems

R. E. K. Stirewalt, Reimer Behrends, Laura K. Dillon
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
e-mail: {stire, behrends, ldillon}@cse.msu.edu

1 Introduction

It is well known that the expressive power afforded by the use of concurrency comes at the expense of increased complexity. Without proper synchronization, concurrent access to shared objects can lead to race conditions, and incorrect synchronization logic can lead to starvation and/or deadlock. Moreover, concurrency confounds 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.

Nowhere is this complexity more evident than in multi-threaded applications in which concurrent threads operate on shared data. Whereas parallel and distributed systems are becoming increasingly important, the safe and reliable use of concurrency in multi-threaded shared-memory systems has emerged as a fundamental and pervasive engineering concern, similar in nature to safety, efficiency, and scalability.

In prior work [4,5] we developed a powerful model of *synchronization contracts* for object-oriented languages that addresses this complexity. In lieu of writing low-level code to acquire and release shared objects, programmers declare synchronization contracts in a module's interface. The contracts declare the circumstances under which the modules in a program require exclusive use of shared objects in a form that permits automated inference of how to synchronize processes in using these objects. A distributed run-time scheduler negotiates the contracts on behalf of processes, ensuring that the contracts of all modules are met while simultaneously guard-

ing against data races and deadlocks that can feasibly be avoided. Separating concurrency concerns from functional code simplifies programming, and localizing concurrency concerns in module interfaces simplifies reasoning.

We have incorporated this model into two languages, Ruby [31] and Eiffel [23].¹ Currently, we are integrating it with C++ and are investigating how synchronization contracts can be used synergistically with traditional engineering-design methods and formal-analysis techniques to provide stronger safety and reliability guarantees.

This paper provides an introduction to our synchronization-contract model and outlines current research activities aimed at enabling technology transfer. We begin by providing background on synchronization contracts and on different approaches for supporting concurrency in modern programming languages (**Section 2**). We then describe our contract model (**Section 3**) and highlight the model's key benefits (**Section 4**). Finally, we outline ongoing and future research directions (**Section 5**).

2 Background

This section explains the rationale underlying the general notion of synchronization contracts (**Section 2**). It then provides background on alternative synchronization mechanisms, which modern programming languages provide to support multi-threading. This background helps in understanding the strengths and limitations of different approaches and, in particular, of our synchronization-contracts model.

¹the extended Eiffel compiler is available at: <http://www.cse.msu.edu/~behrends/universe>.

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 (**Section 2.2**) and explicit message passing (**Section 2.3**). We discuss the latter for completeness, but only briefly since our model is designed expressly for shared-memory systems.

2.1 Synchronization contracts

Synchronization contracts extend Meyer’s *design by contract* approach [24] to address the safety and reliability concerns of concurrent and distributed systems [7]. Briefly, the term *contract* refers to a formal agreement between a software module, called the *supplier*, which provides a service, and other modules, called *clients*, which 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 principle that Meyer calls “guarantee more by checking less” [24]. For example, a contract-aware module will not contain code to check its pre-condition, as such would be redundant with its contract. Moreover, explicit contracts facilitate modular reasoning about correctness via so-called assume-guarantee arguments. Consequently, contracts enable the design of efficient service implementations, reasoning about the correctness and reliability of module interactions, and compile-time optimizations.

A synchronization contract describes client and supplier rights and responsibilities when performing operations that use 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. Meyer proposed a contract model, called SCOOP [24], in which operation pre-conditions are interpreted as guards, such that an operation invocation will block until its pre-condition is satisfied. Thus, unlike the pre- and post-condition style contracts, which must be verified at design time, synchronization contracts are *negotiated* at run time.

Models, such as SCOOP, assign much of the responsibility for contract negotiation to a run-time system, which schedules processes based on deadlock- and starvation-avoidance heuristics. Commensurate with the “guarantee more by checking less” principle, synchronization contract-aware modules are simpler and more reliable than their contract-unaware equivalents because the ability of the run-time system to negotiate contracts obviates the need to implement tricky synchronization logic in a module’s code.

2.2 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 [25], 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.

By contrast, a *general race* occurs when sequences of events by multiple threads execute non-deterministically in such a way that different interleavings of their actions might produce different results. A classic example of a general race occurs in statements such as:

```
if not channel.full() then
  channel.put()
end
```

(1)

where we assume all methods of `channel` execute atomically. Because these methods execute atomically, there can be no data races on `channel`. However, if two producer threads T_1 and T_2 are executing (1) concurrently, in some interleavings T_2 can cause `channel.full()` to change from `false` to `true` after T_1 has checked the condition but before T_1 has executed `channel.put()`.

Shared storage approaches typically 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 [17]; whereas client-side

mechanisms—e.g., synchronized blocks in Java, lock statements in Sather [12], holdif statements in CEE [20], procedures with separate arguments in SCOOP, and non-blocking transactions in Java [15]—trace back to *conditional critical regions* [14]. Supplier-side mechanisms have the advantage that they easily protect against data races. For example, if `channel` 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 the supplier with code that acquires and then releases a lock [8].

On the other hand, difficulties arise when using supplier-side synchronization mechanisms to coordinate interactions among multiple suppliers and prevent 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. These solutions require the supplier developer to anticipate 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 [8, 18, 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, maintaining history information in the supplier can lead to supplier-side inheritance anomalies [18, 22, p. 52].

2.3 Message passing approaches

The *message passing* 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. 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* [1, 26], which queues client requests for services and processes the queued requests in a local event loop.²

The message passing 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 the message passing approach are subject to general races and history sensitivity [8, 18].

Moreover, some problems are most naturally modeled by multiple processes operating on shared data. Solutions to such problems using messaging typically require a new process for each shared object, and so incur performance penalties. The Ada-95 rationale cites this problem as one of the reasons for introducing *protected types*.³

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 [29], the main problem for a program that uses active objects and messaging is that of coordinating these events. Modern applications often integrate heterogeneous components, which may use different synchro-

²Such an event loop is often called a *body*. For a good discussion of the design and implementation of active objects, see [9].

³Other 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 “but of line with a modern object-oriented approach” [19, § II.9].

nization paradigms, so that neither approach excludes the other.

3 Our Model

Our contract model provides for client-side concurrency management in the shared storage model. This section describes the key concepts needed to understand this model (**Section 3.1**) and provides two simple examples. To be concrete, we couch the examples in our Eiffel extension. We use the first example to illustrate the negotiation of synchronization contracts during execution of a program (**Section 3.2**) and the second to show how local synchronization contracts compose to achieve farther reaching effects while still allowing local reasoning (**Section 3.3**).

3.1 Concepts and definitions

Contracts in our model take the form of data invariants, called *concurrency constraints*, and are associated with modules that encapsulate cohesive object structures, called *synchronization units*. Under this model, processes operate in disjoint data spaces, called *realms*, which expand and contract over the lifetime of a program in order to simultaneously satisfy the contracts of each synchronization unit in each process. A run-time system arbitrates the inter-process negotiation of these contracts by *migrating* synchronization units among the realms of the competing processes in order to satisfy the concurrency constraints that pertain to these units. The runtime system also schedules or suspends processes to guarantee that realms contain the necessary synchronization units while ensuring the realms remain disjoint.

To illustrate how synchronization units and concurrency constraints are created and used, consider the example in **Figure 1** which is written in a version of Eiffel that has been extended to support our contract model. An instance of class CHANNEL_MONITOR listens for events over a channel and writes high priority events to a display. The key word *synchronization* signifies that each instance of CHANNEL_MONITOR defines a *root* object for a new synchronization unit. This synchronization unit is to hold all objects that are to be accessed solely through the root channel monitor object. For example, each CHANNEL_MONITOR object encapsulates its own DISPLAY object (created in

```
(1) synchronization class CHANNEL_MONITOR
(2) creation
(3)   make          -- constructor
(4) feature { NONE } -- private attributes
(5)   -- a unit variable
(6)   channel: CHANNEL
(7)   -- a typestate variable
(8)   reading: BOOLEAN
(9)   display: DISPLAY
(10) feature { ANY }
(11)   make(a_channel: CHANNEL) is
(12)   do
(13)     channel := a_channel
(14)     create display
(15)     reading := false
(16)   end
(17) feature { ANY }
(18)   start is      -- process entry point
(19)   local event: EVENT
(20)   do
(21)     from until false loop
(22)       reading := true
(23)       event := channel.pop
(24)       reading := false
(25)       if event.priority >= 100 then
(26)         display.show(event)
(27)       end
(28)     end
(29)   end
(30) concurrency
(31)   reading => channel when channel.hasData
(32) end
```

Figure 1: Example synchronization class

line 12), which is therefore placed in the synchronization unit defined by the CHANNEL_MONITOR object.

The instance variable `channel` holds a reference to the channel being monitored, which is passed to the constructor of the CHANNEL_MONITOR object (line 9). This reference is called a *unit reference* because it references a root instance of another synchronization unit.⁴

The boolean variable `reading` encodes an abstract state property, or a *typestate* [11, 30] of a CHANNEL_MONITOR object.⁵ Intuitively, `reading` is true if and only if the CHANNEL_MONITOR object is extracting an event from the channel. Our current implementations of the contract model require a designer to write code to explicitly represent and maintain typestates over the course of a running computation. In ongoing research, we are investigating more abstract mechanisms for defining typestates.

Concurrency constraints are introduced by the keyword `concurrency`. Informally, concurrency constraints form a subset of propositional formulas, whose atomic propositions can be boolean variables that encode typestates, such as `reading`, and unit variables, such as `channel`. Like any propositional formula, a concurrency constraint has a semantic interpretation that can be either true or false for a given state of the program. A boolean variable has its normal interpretation. A variable referencing a synchronization unit is interpreted as true if and only if the referent is in the realm of the current process or the variable contains a null reference. We say a realm is *complete* if all concurrency constraints associated with a synchronization unit contained in the realm are satisfied and, moreover, if removing any non-root object from the realm would result in some constraint being violated. A process can run only when its realm is complete, meaning that its synchronization contracts have been successfully negotiated. When a process' realm cannot be completed, the process blocks until the synchronization units needed to complete the realm become available and the run-time system migrates them into the process' realm.

For example, the concurrency constraint in line 29 of **Figure 1** prescribes that if `reading` is true then the unit referenced by `channel` must be part of the realm of the process. The qualifier (when `channel.hasData`) further prescribes that the unit referenced by `channel` may be migrated

⁴CHANNEL is a synchronization class; for brevity, we omit its declaration.

⁵In our earlier papers, we refer to boolean variables that are used to represent typestates as *condition variables*.

into the realm of the current process only when `channel.hasData` is true.⁶ Observe that if `reading` is false, the concurrency constraint is trivially satisfied, and no migration is necessary to satisfy it.

Contract negotiation (or re-negotiation) is triggered by assignments to variables that occur in concurrency constraints. Lines 20–22 show an example of a CHANNEL_MONITOR receiving an event from `channel`. When `reading` is set to true in line 20, the process must renegotiate its synchronization contracts. The concurrency constraint on line 29 causes the unit referenced by `channel` to be migrated into the realm of the current process, if that unit is available and it has data to be transmitted; otherwise, the process blocks until such time as the run-time system completes its realm.

A concurrency constraint expresses a designer's intention for a client to access a supplier only when the client is in some particular typestate, and may additionally restrict migration to depend on the typestate of the client. In migrating the client, the run-time systems enforces that the client is in the designated typestate. However, it is up to the designer to write code that conforms to the specified intention; we define as *erroneous* any accesses that do not. For example, if line 20 in **Figure 1** had been omitted, it would be erroneous for a process to execute line 21 because the unit referenced by `channel` would not be in its realm. To avoid such erroneous accesses, our compiler prefixes each method call on a remote synchronization unit with a check that raises an exception if the unit is not part of the current realm. Because a synchronization unit cannot be accessed without being in the realm of the current process, data races cannot occur.

3.2 Realm dynamics

To illustrate how realms grow and shrink in order to satisfy concurrency constraints, **Figure 2** shows a series of UML instance diagrams representing global states at different stages during execution of an example program. The example program creates a SOURCE process and two CHANNEL_MONITOR processes. We depict realms 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

⁶CHANNEL maintains the typestate variable `hasData` to signify that it has data to transmit.

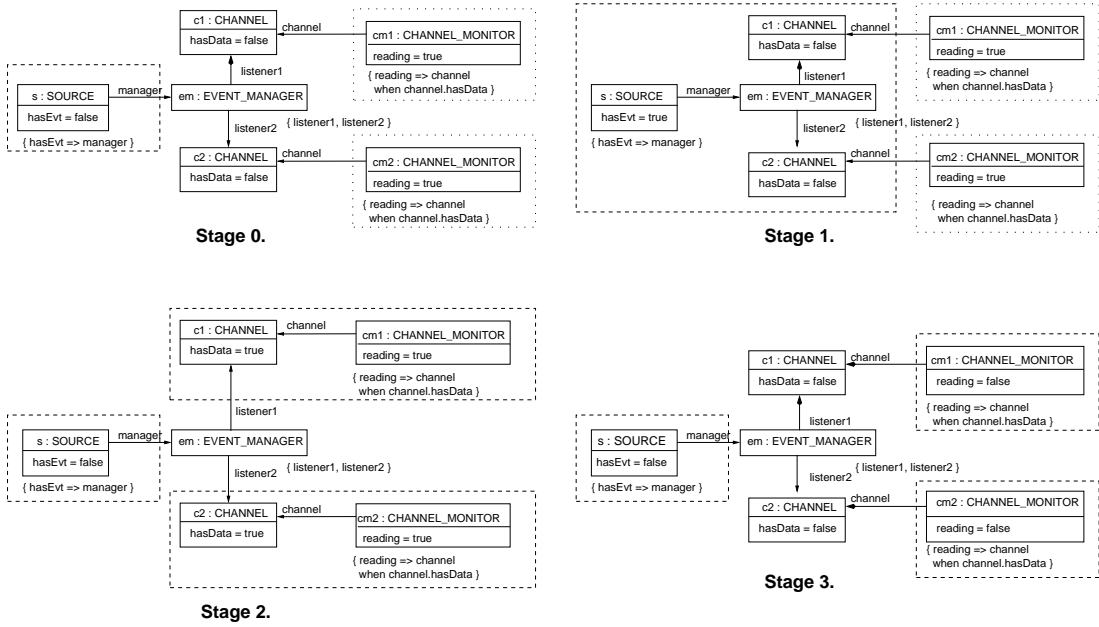


Figure 2: Realm dynamics

process is enabled (resp. blocked). These processes operate over a total of six synchronization units—one SOURCE unit, one EVENT_MANAGER unit, two CHANNEL units, and two 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.

We show unit variables as labels on references (arrows) connecting synchronization units. For example, the SOURCE unit declares one unit variable, `manager`, which references the EVENT_MANAGER unit. Typestate variables are indicated as boolean valued attributes. Thus, the SOURCE unit declares one typestate variable, `hasEvt`. Intuitively, `hasEvt` is true if the source has an event that it can transmit; and false otherwise. We indicate concurrency constraints in curly braces below or alongside the respective synchronization units. For instance, the concurrency constraint of the SOURCE unit is

$$\text{hasEvt} \Rightarrow \text{manager},$$

signifying that a process executing this unit needs exclusive access to the manager when it has an event to transmit. The EVENT_MANAGER unit receives events and broadcasts them to the two CHANNEL units. It has two concurrency constraints, `listener1` and `listener2`. As neither constraint is guarded by a boolean expression, the constraints assert that the event manager requires ex-

clusive access to both of the CHANNEL units referenced by `listener1` and `listener2` in order to execute. The CHANNEL_MONITOR units follow the description in **Section 3.1**, reading events from the channels. The channels do not declare any concurrency constraints.

Initially, the realms of all processes contain just the root instances of their respective synchronization units, as shown in **Stage 0** of **Figure 2**. In addition, the `hasEvt` variable is false, indicating that the source does not have an event to send; both `hasData` variables are false, indicating that the channels do not contain data; and both `reading` variables are true, indicating that the channel monitors need to read data from their respective channels. The concurrency constraint of the source is trivially satisfied, and so it is enabled; whereas the constraints of the event monitors are not satisfied. Moreover, the realms of the event monitors cannot be completed by migrating synchronization units—the respective channels must first assign true to `hasData`. Thus, the processes that own the CHANNEL_MONITOR units are blocked.

Stage 1 of **Figure 2** begins when the source generates an event and assigns `hasEvt` the value true. Its realm expands to include the event manager, in order to satisfy the constraint associated with the source, and both channels, in order to satisfy the constraints associated with the event manager. All three units are migrated as a single atomic operation.

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. Algorithms that implement a distributed runtime system capable of negotiating concurrency constraints are described in [4].

3.3 Contract composition

The previous example illustrates how a local constraint between the source and the event manager has farther reaching effects due to the propagation of concurrency constraints. In **Stage 1** of **Figure 2**, 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 a single atomic operation. This interpretation results from composing contracts by conjunction and accounts for both the modularity of synchronization contracts and their ability to express complex collaborations. In this section, we show how the composition of synchronization contracts addresses the *layering problem*, which arises when using layered architectures.

The layering problem is illustrated in **Figure 3** by the collaboration between the clients, $C1$ and $C2$, the suppliers, $S1$ and $S2$, 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 $C1$ (resp. $C2$) 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 $S1$ (resp. $S2$) 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.

4 Key Benefits

Our contractual approach provides a number of benefits to the software designer. First, as contracts, concurrency constraints separate concurrency concerns from the computational code, while assigning clear rights and responsibilities. For example, consider the constraint `hasEvent => manager` associated with the source unit in **Figure 2**. This constraint allows the designer to safely assume that, once the source unit assigns `hasEvent` the value true, the process executing the unit will have exclusive access to the event manager until the unit next assigns `hasEvent` the value false. Moreover, the constraint also signals that the source unit may assume that `hasEvent` is true when it acquires access to `manager`.

In addition to separating concerns, contracts raise the level of abstraction of concurrent programming, obviating the need to intersperse low-level synchronization instructions in procedural code. In the previous example, notice that without contracts, a designer would have to assign the responsibility for ensuring exclusive access to either the instance of `SOURCE` or the instance of `EVENT_MANAGER` referenced by `manager`. Consequently, the “functional logic” in one or both of these modules would be interspersed with low-level logic for acquiring and releasing locks. By itself, this logic can be tricky to implement, especially when objects are involved in transitive exclusion dependencies, such as the dependencies between the instance of `SOURCE` and the two instances of class `CHANNEL`. This complexity only increases when the logic is interleaved with the functional logic.

Concurrency constraints also compose cleanly to allow for the propagation of synchronization contracts needed for complex collaborations. Thus, the

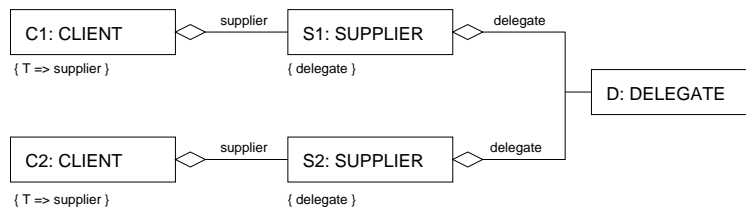


Figure 3: The Layering Problem

concurrency constraint of the source in **Figure 2** is implicitly composed with the concurrency constraints of the event manager, which the source references through the `manager` link. This feature allows the modularization of concurrency constraints: Note that the client (i.e., the source) need not be aware of the supplier’s (i.e., the event manager’s) synchronization requirements. Because the runtime system handles negotiation of composed contracts while preventing avoidable deadlock and starvation, complex design regimes, such as those cataloged in [21], are not needed to manage safety and liveness concerns in large systems.

Of course, implicit propagation of local constraints 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 more primitive 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 should permit development of compositional static analysis algorithms to efficiently determine that a larger class of programs is free from deadlock.

Another benefit of synchronization contracts is their use in documentation and verification. Each module’s concurrency constraints are expressed declaratively in terms of the module’s tpestates and those of its suppliers. Concurrency concerns thus become part of the module’s interface. The ability to reason about the concurrent interaction of modules by inspecting their interfaces allows a software designer to minimize the synchronization logic of a program. Minimizing synchronization logic is important in concurrent programming, not only because it simplifies the design, but also because spurious synchronization instructions can lead to deadlock.

Finally, synchronization contracts also introduce a useful degree of redundancy that guards against many common programming errors. For example, the omission of either the concurrency constraint in line 29 of **Figure 1** or the assignment in line 20 would result in a runtime exception instead of a race condition. Our model thus automatically guards against data races. Furthermore, it also protects against a number of general races.⁷

A potential drawback of our existing model of synchronization contracts is the limited expressive power of contracts. For example, our synchronization contracts do not yet provide functionality where multiple readers can access the same object concurrently, or where multiple methods on the same object can be called concurrently if they do not interfere with one another. Noble, Holmes, and Potter present a model of mutual exclusion contracts that allows such *intra-object concurrency*, but this model is intended to be used as a design tool rather than as a language extension [27]. Also, while our runtime system avoids or recovers from deadlocks whenever feasible, a programmer can declare synchronization contracts that can lead to deadlock. We believe it should be possible to complement the deadlock avoidance/recovery algorithms used during run-time contract negotiation with static deadlock analysis to guarantee that a program will not deadlock.

5 Ongoing Research

To date, we have taken several steps to validate the efficacy of our approach. We have currently extended two languages, Eiffel and Ruby, with support for synchronization contracts and have developed a robust extended-Eiffel compiler. This compiler generates code that exploits an optimized run-time system, whose performance scales well with the number of synchronization units [4]. More recently, we used

⁷See [4] for details.

the extended-Eiffel compiler to develop a large case study, a multi-threaded web server designed around the Apache architecture [6]. We subjected the design to several maintenance tasks to judge how contracts support extension and contraction and to begin to understand the patterns and idioms that emerge when applying contracts to a large design.

We believe the identification of these patterns and idioms to be crucial to tech transfer, given the “solutions” that have had an impact on the practice of concurrent/distributed programming. The most widely used solutions come from designs that have proved useful in practice and that have been generalized and recorded in a form that facilitates reuse, e.g., design patterns [13, 28] and handbooks [21]. Solutions in this genre often take the form of elided artifacts, e.g., structural or behavioral diagrams, and heuristic design methods. Unfortunately, code patterns and diagrams are rarely formal enough to support rigorous analysis and verification, and their utility is often limited by what is expressible using the low-level concurrency primitives available in the language used to implement the artifacts. Because synchronization contracts are much more abstract than these low-level primitives, our approach should not suffer from this drawback.

In addition to understanding how our approach raises the level of abstraction in design, we are investigating how, by virtue of its formality, our approach enables analysis and verification. We are particularly interested in seeing how contracts can inform and enable the automatic derivation of finite-state models from source code, such as is provided by tools like Bandera [10], Java Pathfinder [16], and SLAM [2, 3]. One benefit of our contract model is the separation of concerns it affords. For example, we recently developed a language-neutral formal semantics of contract negotiation, independent of the “core computation” that is running under the influence of these contracts. It is an open question as to whether this separation of concern can dramatically improve efficiency or scalability of code-based analysis.

Another open question is whether the formality that enables analysis may prove to be an obstacle to adoption by practitioners. Novel programming-language paradigms based on formal models seem to have trouble finding their way into practice. The new paradigm requires a new way of thinking about a problem, and practitioners have difficulty translating their existing “engineering knowledge” into the new paradigm. Judging from what has impacted practice, practitioners learn better from real computing arti-

facts, even when these artifacts are abstracted into patterns or informal diagrams. If this is true, then a prerequisite to the adoption of an approach like synchronization contracts is a wealth of artifacts that use them and heuristic methods to guide their application. Moreover, these artifacts must be organized into patterns, idioms, and visual diagrams, and they must be accompanied by design methods that employ formal analysis and verification of safety properties.

Acknowledgements Partial support for this research was provided by the Office of Naval Research grant N00014-01-1-0744 and by NSF grants EIA-0000433 and CCR-9901017.

References

- [1] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.
- [2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, 2001.
- [3] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *Proc. of the ACM SIGPLAN Symposium on the Principles of Prog. Languages*, 2002.
- [4] R. Behrends. *Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages*. PhD thesis, Michigan State University, East Lansing, Michigan USA, December 2003.
- [5] R. Behrends and R. E. K. Stirewalt. The Universe Model: An approach for improving the modularity and reliability of concurrent programs. In *Proc. of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'2000)*, 2000.
- [6] R. Behrends, R. E. K. Stirewalt, and L. K. Dillon. Avoiding serialization vulnerabilities through the use of synchronization contracts. In *Proc. of the Workshop on Specification and Automated Processing of Security Requirements*, 2004.
- [7] A. Beugnard et al. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

- [8] T. Bloom. Evaluating synchronisation mechanisms. In *Seventh International Symposium on Operating System Principles*, pages 24–32, 1979.
- [9] J. P. Briot, R. Guerraoui, and K. P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3), 1998.
- [10] J. C. Corbett et al. Bandera: Extracting finite-state models from java source code. In *Proc. of the Int’l Conf. on Software Engineering*, 2000.
- [11] R. DeLine and M. Fähndrich. Typestates for objects, 2004. Submitted to ECOOP’04.
- [12] C. Fleiner and M. Philippsen. Fair multi-branch locking of several locks. In *International Conference on Parallel and Distributed Computing and Systems*, pages 537–545, Washington D.C., 1997.
- [13] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison–Wesley, 2003.
- [14] P. Brinch Hansen. Structured multi-programming. *CACM*, 15(7):574–578, July 1972.
- [15] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–402. ACM Press, 2003.
- [16] K. Havelund and T. Presburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), 1998.
- [17] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.
- [18] D. Holmes. *Synchronisation Rings: Composable Synchronisation for Object-Oriented Systems*. PhD thesis, Macquarie University, Sydney, 1999.
- [19] Inc. Intermetrics. Ada 95 rationale: The language and the standard libraries. Online version available at: <http://www.adahome.com>.
- [20] G. Jalloul and J. Potter. A separate proposal for Eiffel. In *Proceedings of the 2nd Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific ’93)*. Prentice Hall, 1993.
- [21] D. Lea. *Concurrent Programming in Java*. Addison–Wesley, second edition, 2000.
- [22] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, Cambridge, MA, 1993.
- [23] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1997.
- [24] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
- [25] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.
- [26] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.
- [27] J. Noble, D. Holmes, and J. Potter. Exclusion for composite objects. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA’2000)*, 2000.
- [28] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2 of *POSA*. Wiley & Sons, 2000.
- [29] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.
- [30] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12:157–171, 1986.
- [31] D. Thomas and A. Hunt. *Programming Ruby*. Addison–Wesley, 2000.