Contract-based Synchronization of Multi-threaded Java Programs

By

Yi Huang

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Computer Science

2011

ABSTRACT

Contract-based Synchronization of Multi-threaded Java Programs

By

Yi Huang

Today, most new software products use concurrency in some capacity. However, 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 or deadlock. Moreover, concurrency confounds the development of reusable software modules because code implementing an application's synchronization logic tends to be tightly interleaved with the "functional" code. Interleaving complicates program understanding and maintenance, producing brittle applications.

Contract-based models of synchronization, such as the Synchronization Units Model (Szumo), attempt to address these problems by: (1) expressing synchronization concerns in declarative synchronization contracts to separate them from functional code; and (2) using a runtime system that dynamically interprets and "negotiates" the contracts, thereby automatically synchronizing threads. However, this approach requires a special compiler and runtime system, making it difficult to integrate Szumo into mainstream object-oriented programming languages or conduct empirical studies to understand software engineering tradeoffs when using Szumo or hand coding synchronization.

This thesis investigates two "lighter-weight" approaches for integrating a contract-based synchronization model with a mainstream object-oriented programming language. The first approach works with any multi-threaded Java program. In this approach, an application programmer adds special Java annotations to a class whose methods contain only functional code. A compiler plugin generates synchronization code from the annotated program based

on synchronization concerns declared in the annotations; the generated synchronization code is added to the annotated program, which then executes in a standard JVM. The second approach targets IP telecommunication (IPT) services that are deployed to a SIP servlets container. It makes use of a synchronization middleware. Instead of embedding synchronization code in the message handlers that implement a service, a programmer provides a synchronization contract that is loaded when the service is deployed to a container running our middleware. The middleware intercepts messages that a container routes to the service and consults the contract to determine when to schedule the message handler thread.

Contributions of work reported in this thesis include:

- Development of a generative approach that permits use of synchronization annotations with a mainstream object-oriented language.
- Development of a middleware approach that permits use of contract-based synchronization with a standard execution platform for services.
- Demonstration that contract-based synchronization enables packaging the implementation of synchronization as an OTS component, which can be seamlessly swapped with one that implements a different protocol, e.g., to tune performance.
- Presentation of results of case studies with both approaches.

The work described in this thesis was performed in collaboration with AT&T Research Labs and Oracle Research Labs.

ACKNOWLEDGEMENTS

An amazingly recurrent pattern in my life is that I always end up with wonderful people. The journey of my Ph.D. pursuit is not an exception.

It started in Fall 2006, when I was lucky to meet Professor Dillon, who is responsible, diligent and very well-respected, and Professor Stirewalt, who is technical, intelligent and very knowledgeable. It has been a great pleasure to work with them since then. I would not be able to finish this thesis without their valuable guidance, which has been way beyond the extent of academia.

Later, I met very talented and supportive researchers at AT&T Research Labs and at Oracle Research Labs, including Gregory Bond, Eric Cheung, Peter Kessler, David Leibs, Tom Smith, Michael Van De Vanter, Mario Wolczko, Pamela Zave, et. al. During our exciting collaborations, we have established close friendships that will last forever.

Of course, nothing is meaningful to me without my lovely wife, Yuan. I would not be able to finish this journey without her brave sacrifice, her tremendous support, and the precious surprise she gave me in the end of the journey.

There are also many other wonderful people, who have made indispensable and positive impact on my life. I sincerely cherish and appreciate their being around.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Today, most new software products use concurrency in some capacity. The trend is toward systems that are increasingly interactive and distributed, even in high-assurance application domains. Examples include high-demand information systems, interactive command and control systems, and many embedded systems.

However, 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 protocols and decisions are difficult to localize into a single software module. 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.

The fundamental problem is a tendency of synchronization logic to be tightly interleaved with the "functional" code. Interleaving complicates program understanding and mainte-

nance, producing brittle applications [59]. Methods combat the interleaving problem using tools, processes, and artifacts that support separation of concerns. Unfortunately, synchronization logic is a global and cross-cutting concern, which is difficult to implement and is resistant to clean separation from the primary functional logic of programs.

Previously, we developed a compositional model of synchronization called the Synchronization Units Model (Szumo) to address the safety and extensibility concerns that arise in multi-threaded programs [6, 63]. Szumo trades generality for compositionality by focusing on the category of strictly exclusive systems, in which multiple threads compete for exclusive access to dynamically changing sets of shared resources. This narrowing of focus was motivated by the observation that many applications fit well in this category and that, in such cases, we can exploit a clean, compositional model of design and verification [61, 14].

In Szumo, threads contend with one another for exclusive access to sets of synchronization units. Intuitively, a synchronization unit is an application-defined indivisible unit of sharing—at any time during execution, a thread is allowed to access either all of the objects contained in a synchronization unit or none of the objects contained in that unit [5, 63]. In lieu of embedding code that invokes OS-level synchronization primitives, a synchronization unit declares one or more synchronization constraints, which specify the conditions under which the unit—acting as a client—needs exclusive access to units—acting as the client's direct suppliers—that the client holds references to. Transparent to the programmer, as the values of these conditions change during execution, threads negotiate for exclusive access to suppliers in accordance with the constraints declared by any clients that they are executing. The declarative nature of synchronization constraints and this automated process of negotiation are key to Szumo's support for extension and maintenance [7].

Using a language extension to Eiffel for directly expressing the features of a Szumo design, we were able to show how Szumo supports modular descriptions of the synchronization concern, raising the level of abstraction at which synchronization concerns are expressed,

and automating implementation and enforcement of these concerns [5, 63]. A case study involving maintenance and evolution of a web server, based on the architectural design of Apache, demonstrated that a well-designed Szumo application could be safely modified and extended without requiring redesign of the original synchronization concern [7].

However, the choice of Eiffel as a base language and the decision to build support for synchronization directly into the language implementation (i.e., the compiler and run time system) made it difficult to evaluate the practical value of Szumo. We chose Eiffel as a base language because it is a powerful, but concise, pure object-oriented language with a clean semantics. But while this choice allowed us to quickly develop a reference implementation of Szumo, the use of a programming language with a relatively small user base made it difficult to find suitable benchmark programs for purposes of comparison. In addition, the need to use Eiffel was a barrier to others using Szumo/Eiffel for developing their own applications.

The uniform syntax and logical coherence of Eiffel's language features also made implementing a compiler and run-time system that interpret and enforce synchronization contracts relatively straightforward. But by tangling the implementation of synchronization with other complex run-time concerns, this decision made it difficult to separate out effects on performance of different synchronization algorithms. The synchronization algorithm implemented in the Szumo/Eiffel run time was designed to admit more concurrency than other, better known algorithms, e.g., those based on resource numbering or a gatekeeper. However, the extra concurrency comes at a cost in complexity and additional overhead. Unfortunately, to evaluate tradeoffs of using different protocols in practice would require rewriting the Szumo/Eiffel run time. Additionally, our experiments showed that a web server based on the same design as the Apache Web Sever, but written in Szumo/Eiffel, was not nearly as efficient in its use of locks as a commercial Apache server. While we believe that the use of locks in Szumo/Eiffel can be optimized, we do not have the resources to optimize our language implementation to be competitive with a commercial tool.

This thesis investigates methods for integrating Szumo with mainstream object-oriented programming languages and domain-specific execution platforms in a fashion that (1) works with existing language implementations and platform infrastructures and (2) cleanly separates out the synchronization protocol used for resolving contention. Criteria (1) permits development of applications with Szumo using commercial compilers, platforms, and software design environments. Criteria (2) allows tuning, or even replacing, the synchronization protocol without affecting the functional code.

Specifically, we investigate two approaches for integrating Szumo with Java. The first is a general approach, which works with any multi-threaded Java program. Briefly, an application programmer embeds synchronization annotations into a Java program that implements only functional logic, and which is therefore said to be synchronization agnostic. Intuitively, the synchronization annotations indicate the shared resources a thread needs to use and the conditions that a thread expects to be true when executing methods of the synchronization-agnostic program. The programmer then uses a compiler plugin to weave synchronization code into the synchronization-agnostic program. The automatically generated code synchronizes threads as they invoke methods so as to prevent data races and avoid deadlock. We refer to this approach as the generative approach because it generates a new Java program from the synchronization-agnostic program; the new program is then compiled and executed with any Java Virtual Machine (JVM).

The second approach for integrating Szumo with Java targets IP telecommunication (IPT) services that are deployed to a SIP servlets container, the industry standard in this domain [40]. The approach makes use of a synchronization middleware instead of weaving synchronization code into an existing program. Briefly, an application in this domain, commonly called a service, responds to messages it receives from asynchronous clients (e.g., SIP devices and other services). The container simplifies programming the service by automating details of message routing and transmission; creation, dispatch, and termination

of a thread to handle each incoming message; and other non-functional concerns. The application programmer writes servlets, Java classes whose methods define the behaviors of message handlers. Because synchronization has proven difficult to relegate to the container, in current practice, a programmer may need to mix functional code with synchronization code in a message handler. However, with our middleware approach, instead of embedding synchronization code in the message handlers, the programmer provides a separate synchronization contract, which indicates the shared resources a handler needs to access. When a service is deployed to a SIP servlets container that is configured to use our synchronization middleware, the middleware automatically loads the synchronization contract. Subsequently, it automatically intercepts each message that the container routes to the appropriate message handler; uses the contract to determine and then acquire the resources needed before invoking the the servlet; and releases the resources when the handler returns.

To satisfy criteria (2), both of these approaches utilize a library of negotiator components, which encapsulate alternative synchronization protocols. These components can be used interchangeably with both approaches to resolve contention for shared resources while avoiding deadlock and starvation. Concurrency experts can augment this library with custom negotiators by implementing a negotiator API.

In sum, this thesis makes the following contributions:

- We show that a generative approach permits use of contract-based synchronization with a mainstream object-oriented language and with standard programming tools and development environments.

- We show that a middleware approach permits use of contract-based synchronization with a standard service execution platform.

- We show that contract-based synchronization permits separating the implementation of the synchronization protocol used by a multi-threaded program from the implemen-

tation of other runtime concerns, thereby facilitating modification and tuning of the synchronization protocol.

- We present results of case studies with both approaches. Those using the generative approach show that it allows elegant solutions for many classic synchronization problems. Because work on the middleware approach is more mature, one of the case studies performed with this later approach also assesses performance; it shows that the middleware approach can scale for use in implementing realistically complex IPT services.

This thesis is organized as follows. Chapter 2 uses an example to describe the well-known problem in implementing thread synchronization. Chapter 3 describes the related work that aims to address this problem; and discusses how it inspires the design of our contract-based synchronization approach. Chapter 4 focuses on the difference between our synchronization approach and its precedent model, Szumo. Chapter 5 describes our generative approach to integrating synchronization contracts with Java language. Chapter 6 describes our middleware approach to integrating synchronization contracts with a standard execution platform in the IPT domain. Chapter 7 concludes this thesis. Chapter 8 discusses future work.

Chapter 2

The Problem: Implementing Thread Synchronization

The need to properly synchronize concurrent threads that access shared objects greatly complicates development of multi-threaded applications. Failing to synchronize thread activities that concurrently access a shared resource can result in data corruption. However, too coarsely synchronizing these activities can degrade concurrency, whereas too finely synchronizing these activities can give rise to deadlocks or starvation. Unfortunately, a suitable synchronization protocol is hard to reason about due to explosion in the number of possible orderings of event occurrences. Moreover, nondeterminism in this ordering also makes errors in multi-threaded applications extremely hard to detect and replicate. Without proper abstractions to facilitate separation of concerns, a programmer must mix synchronization code with primary functional code in the implementation of a multi-threaded application. This practice obscures the programmer's intent and produces code that is hard to understand and maintain.

This chapter illustrates well-known issues in implementing thread synchronization. It presents a simple but representative example involving both mutual exclusion and condition

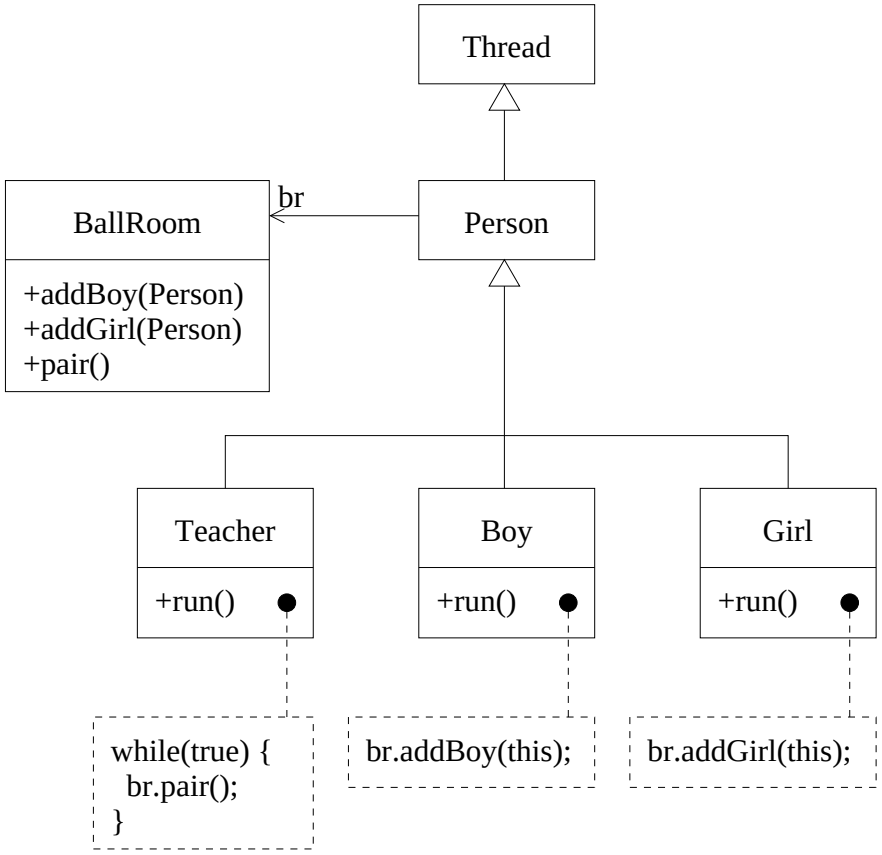


Figure 2.1: Class diagram of the ball room example

synchronization [51, 17, 15].

2.1 Ball room example

We give an overall design of the example (Section 2.1.1) and then describe its implementation (Section 2.1.2).

2.1.1 Design

The program coordinates three types of objects—girls, boys, and teachers. Each executes on a dedicated thread and shares the same instance of class BallRoom, whose task is to allow a

teacher to pair a boy and a girl who have arrived at a ball room to dance. Synchronization is needed because a child can arrive at any time and because a teacher must wait until children of different genders are present to pair them. Figure 2.1 illustrates a class diagram that shows the design of this program. BallRoom models its functional logic using three public methods (listed in the lower part of the box). Person specializes Thread, which is provided as part of the Java SDK, with a reference br to the shared instance of BallRoom. Teacher, Boy and Girl derive from Person, overriding Thread's run() method to call different methods on br, as determined by their roles. The entry point of the program, Main (not shown), creates a finite number of Teacher, Boy, and Girl objects; a thread for each; and a single BallRoom instance. It then configures them as indicated in the figure.

2.1.2 Implementation

We focus on the implementation of class BallRoom. Figure 2.2 illustrates its functional code. The ballroom maintains two queues that its public methods use to simulate arrival of a boy, arrival of a girl, and pairing of a boy and a girl. While this code is straightforward, the synchronization code of this class represents the main source of complexity.

The synchronization code must implement both mutual exclusion and condition synchronization. Mutual exclusion ensures queue consistency by preventing teachers and children from accessing the queues concurrently, as Java's LinkedList is not a thread-safe data structure. In Figure 2.3, mutual exclusion is implemented using synchronized blocks. By embedding queue access (lines 7, 14, 24–25) inside such blocks (lines 6, 13, 20 and 22), the program serializes concurrent threads that attempt to use the same queue and consequently precludes data races. Condition synchronization prevents a teacher from pairing children (i.e., executing pair()) if there are no boys (i.e., queue boys is empty) or no girls (i.e., queue girls is empty). It is implemented using wait() (lines 21 and 23), which suspends a teacher's execution, and notify() (lines 8 and 15), which activates a suspended teacher when a child

```
1  public class BallRoom {
     private Queue<Person> boys = new LinkedList<Person>();
3    private Queue<Person> girls = new LinkedList<Person>();

5    public void addBoy(Person boy) {
       boys.add(boy);
7    }

9    public void addGirl(Person girl) {
       girls.add(girl);
11   }

13   public void pair() {
       boys.remove();
15     girls.remove();
     }
17 }
```

Figure 2.2: Implementation of BallRoom's functional logic

arrives.

2.2 Synchronization problem

Implementing synchronization using Java synchronization primitives is difficult and prone to bugs because of their subtle and implicit semantics. First, lock.wait() and lock.notify() may be called only by the thread that holds lock. Second, a call to lock.wait() implicitly releases lock so that another thread can acquire it, which may activate the former thread in the future. Third, when a call to lock.wait() returns, the calling thread may still have to block because it implicitly attempts to re-acquire lock.

For example, the while loops in Figure 2.3 (lines 21 and 23) are necessary because, when a thread returns from a call to lock.wait(), another thread contending to obtain lock may invalidate the condition that the former thread relies on. Therefore, substituting while with

```
1  public class BallRoom {
     private Queue<Person> boys = new LinkedList<Person>();
3    private Queue<Person> girls = new LinkedList<Person>();

5    public void addBoy(Person boy) {
       synchronized (boys) {
7        boys.add(boy);
         boys.notify();
9      }
     }
11
     public void addGirl(Person girl) {
13     synchronized (girls) {
         girls.add(girl);
15       girls.notify();
       }
17   }

19   public void pair() {
       synchronized (boys) {
21       while (boys.size() == 0) boys.wait();
         synchronized (girls) {
23         while (girls.size() == 0) girls.wait();
           boys.remove();
25         girls.remove();
         }
27     }
     }
29 }
```

Figure 2.3: Complete implementation of BallRoom

if will cause synchronization errors. Another likely error is to juxtapose the synchronized blocks in pair() (lines 20 and 22). They must be nested, because a teacher needs a boy and a girl to be available at the same time.

This example also illustrates the affects of mixing synchronization code and functional code on program comprehension and maintenance. Our implementation of class BallRoom contains more synchronization code than functional code, and the synchronization code is significantly more complicated. The method bodies are hard to comprehend, and the method names do not reflect their synchronization semantics—for instance, a better name would be addBoyAndNotifyTeachers in lieu of addBoy; however, such long method names are either awkward or impossible to come up with in practice. Because synchronization details tend to spread over the entire codebase, a programmer often has to parse and comprehend several method bodies to properly use or maintain such code. Such reasoning is tedious, time-consuming, and error-prone.

Moreover, mixing synchronization code with functional code makes it hard to extend an existing program with new features because it increases coupling between methods. Synchronization hazards are easy to introduce and hard to detect and eliminate, producing brittle programs. Consider, for example, extending class BallRoom with a waiting room used to accommodate children waiting to be paired. Because the size of the waiting room is limited, arriving children must wait outside of the waiting room if it is full. A teacher will need to notify them when space becomes available. Figure 2.4 illustrates an implementation of this extension. The added functional code declares a constant defining the size of the waiting room (line 5) and a new variable (line 6), used to keep track of the number of children who are waiting in the waiting room; and modifies this variable (lines 12, 21 and 31). The added synchronization code introduces a new Java intrinsic lock (line 7), used to serialize concurrent access to numChildrenWaiting; three new synchronized blocks (lines 10, 21 and 30), used to guarantee exclusive access to the waiting room; two while loops (lines 11 and

```
1  public class BallRoom {
     private Queue<Person> boys = new LinkedList<Person>();
3    private Queue<Person> girls = new LinkedList<Person>();

5    private final int SIZE = 10;
     private int numChildrenWaiting = 0;
7    private Object waitingRoomLock = new Object();

9    public void addBoy(Person boy) {
       synchronized (waitingRoomLock) {
11       while (numChildrenWaiting == SIZE) waitingRoomLock.
             wait();
         ++numChildrenWaiting;
13       synchronized (boys) {
           boys.add(boy);
15         boys.notify();
         }
17     }
       }
19   }

21   public void addGirl(Person girl) { ... }

23   public void pair() {
       synchronized (boys) {
25       while (boys.size() == 0) boys.wait();
         synchronized (girls) {
27         while (girls.size() == 0) girls.wait();
           boys.remove();
29         girls.remove();
           synchronized (waitingRoomLock) {
31           numChildrenWaiting -= 2;
             waitingRoomLock.notifyAll();
33         }
         }
35     }
     }
37 }
```

Figure 2.4: Implementation of BallRoom extended with a waiting room

21), used to suspend children in case the waiting room is full; and one notification, used to notify a child that a space is available in the waiting room (line 32).

Although the code in Figure 2.4 looks reasonable, it suffers from four potential deadlock cases. The first two deadlock cases are related to a limitation of Java's synchronized statements. Because a synchronized statement can acquire only one lock at a time, a programmer must nest such statements in order to acquire multiple locks. However, nesting synchronized statements results in incremental locking; therefore, the programmer must pay attention to the nesting order to avoid deadlocks. Unfortunately, due to the mixing of the synchronization code with the functional code, the programmer typically has much less control over the nesting order when she extends an existing code than when she writes a program from scratch. For example, in method addBoy(...), locking waitingRoomLock (line 10) is used to check whether the waiting room is full, while locking boys (line 13) is used to emulate a boy entering the waiting room. Because a boy can enter the waiting room only when the waiting room is not full, the programmer has to lock waitingRoomLock before boys. Likewise, in method pair(...), the programmer has to lock boys (line 25) before waitingRoomLock (line 30), because the former lock is used to check whether there is a boy in the waiting room and the latter lock is used to emulate a boy (and a girl) leaving the waiting room. Because these locks may be acquired in opposite orders, deadlock can occur. Therefore, one of the first two deadlock cases involves these two methods. The other deadlock case is similar but involves addGirl(...) and pair().

While the first two deadlock cases are caused by incremental locking, the last two deadlock cases are related to a flaw in the synchronization logic. Consider what will happen if the waiting room is full of boys; in this case, all teachers will wait for a girl to enter the waiting room; all girls will wait for a teacher to notify them of a vacancy in the waiting room; and all boys will wait for a teacher to pair them with a girl. Deadlock occurs because no thread will be able to make progress. This deadlock case is more subtle than the first two deadlock

cases because it cannot be detected by simply parsing the implementation code to check the nesting orders. Fixing a deadlock like this typically requires a redesign of the synchronization logic rather than simply patching the existing synchronization code. In Chapter 5, we will revisit these four deadlock cases to see how our approach can help avoid them.

Our contract-based synchronization approach capitalizes on separating the synchronization logic from the functional logic in order to address the thread synchronization problem. It introduces a synchronization contract for representing an application's requirements of mutual exclusion and of condition synchronization at a suitably abstract level. For mutual exclusion, the synchronization contract specifies the resources a thread needs to use as the thread executes a method. For condition synchronization, the synchronization contract specifies the conditions that must be true before a thread executes a method. This manner of specification makes it possible to generate the synchronization code and to automate the synchronization logic. Before describing our contract-based synchronization approach, the next chapter discusses some recent work on alternative synchronization primitives and models.

Chapter 3

Related Work

A synchronization protocol is a plan that coordinates the activities of concurrent threads to guarantee that a system satisfies necessary safety and liveness properties [70]. The protocol may work by allowing a thread to proceed only when it is safe to do so, or it may work by rolling threads back to safe states from inconsistent states. The former case is called pessimistic [22, 1], because it pessimistically assumes that an inconsistent state is likely to occur unless you use a prevention strategy. Example pessimistic protocols [72] range from simple ones, such as resource numbering and gatekeeper [29, 25, 38], to more complex ones [6, 18], which are fairer and can permit more concurrency. The latter case is called optimistic [58, 67], because it optimistically assumes that an inconsistent state is unlikely to happen. Optimistic protocols typically pertain to those using transactions, which distinguish from each other in the techniques of detecting conflicts and the policies of resolving conflicts [9, 46, 11, 32, 26].

A synchronization protocol can either be produced manually by a programmer or automatically generated from some form of high-level specifications. In either case, the implementation of a synchronization protocol requires use of synchronization primitives. Typical synchronization primitives are locks, transactions, and events. This chapter describes these

primitives (Section 3.1) and recent research on automatic programming of the synchronization protocols (Section 3.2).

3.1 Synchronization primitives

Programming languages typically implement synchronization primitives in native or external libraries. To use a synchronization primitive, a programmer instantiates the necessary classes and makes calls to the instances created. This section uses popular and representative libraries to describe locks[1], transactions[2], and events[3]. Briefly, locks are the most fundamental synchronization primitives, which can be used to implement both mutual exclusion and condition synchronization. Transactions and events are higher-level abstractions for circumventing the issues that arise from explicitly manipulating locks. Intuitively, transactions can emulate the effect of mutual exclusion without strictly enforcing it; whereas events facilitate programming of condition synchronization using a more intuitive API.

3.1.1 Locks

Locks are native synchronization primitives in Java. Every object is associated with an intrinsic lock. To manipulate that lock, a programmer calls wait() or notify() methods on that object or embeds that object in synchronized statements. The wait() and notify() methods are keys to implementing condition synchronization; whereas the synchronized statements are used to achieve mutual exclusion. The Java concurrency library [47] implements more advanced locks. Examples include read-write locks, which permit an object to be accessed by multiple reading threads or by a single writing thread; reentrant locks, which permit a thread to acquire and release the same lock multiple times; and priority locks, which permit

[1] using Java runtime environment (JRE)

[2] using Oracle's dynamic software transactional memory (DSTM2)

[3] using Microsoft's .NET framework for C# language

contending threads to acquire a lock based on their priorities.

Explicitly manipulating locks is prone to errors, especially when multiple locks have to be managed at the same time. Take deadlocks as an example. When implementing mutual exclusion, deadlock is caused by incremental locking, when two or more threads all block on a lock that another blocked thread has acquired. When implementing condition synchronization, deadlock is usually caused by a condition loop, when two or more threads all wait for a signal that must be triggered by another waiting thread. Unfortunately, none of the existing pessimistic synchronization protocols are practical to use in resolving this and other synchronization hazards without overly complicating implementation. For example, the resource numbering protocol imposes a total order on the set of shared objects and requires all threads to acquire shared objects according to this order. Although this protocol is conceptually simple, devising and managing a common order can be extremely difficult and error-prone [2].

3.1.2 Transactions

While locks achieve mutual exclusion by preventing threads from concurrently accessing a shared object, transactions emulate the effect of mutual exclusion while permitting these transactions to concurrently access a shared object. The effect of mutual exclusion is achieved by preserving the serializability of these transactions, which means that the outcome of a concurrent execution of these transactions appears to be the same with the outcome of a serialized execution of these transactions. By abandoning the use of locks altogether, transactions can circumvent the synchronization issues that arise from explicitly manipulating locks [9, 46, 11, 32, 26]. Briefly, a thread embeds operations on shared objects in a transaction and performs these operations on copies of the shared objects, rather than attempting to acquire locks on these objects. When the transaction commits, the thread then attempts to reconcile conflicting updates on the shared objects made by the other concurrent transac-

tions. If none of these transactions modifies any the same shared object, all the transactions will succeed. Otherwise, if multiple transactions access the same shared object and if at least one of these transactions modifies that object, a conflict will occur. In this case, at most one of the transactions can succeed and all the other conflicting transactions must roll back and retry.

Transactions are primarily used in database management systems (DBMS), such as MySQL and DB2. Sun's dynamic software transactional memory (DSTM2) is one of the few software libraries available for implementing general object-oriented applications [31]. Briefly, a programmer defines a sequential data type as a Java interface, which consists of the signatures of that data type's accessors (i.e., getters and setters methods). The programmer then passes this interface to the constructor of a given transactional factory, which creates an anonymous class implementing this interface. This constructed class contains a static method for creating instances of the class. The constructed accessors contain hooks to notify the transaction manager when they are called. During a transaction, the transaction manager monitors state changes of the instances of this constructed class and resolves conflicts automatically.

Figure 3.1 illustrates an implementation of BallRoom using DSTM2. Because the shared objects are queues, the programmer defines an interface PersonQueue, which comprises methods like getSize(), setSize(...), getElements(), and setElements(...). These methods access data pertaining to a queue; the operations that can be performed on a queue are encapsulated in a static class PersonQueueMethods (line 12). Separating these operations from the accessors is required by DSTM2, because the instances of PersonQueue (lines 5–6) are created by the anonymous class queueFactory that DSTM2 constructs (lines 2–3). The body of method addBoys(...) is an infinite loop (lines 9–21) that endlessly retries a transaction until the transaction successfully commits (lines 17–19). Conflicts are captured before a commit, when the transaction manager throws an AbortedException (line 14), and during a commit,

```
1  public class BallRoom {
     static private DSTM2Factory<PersonQueue> queueFactory =
3      DSTM2Thread.makeFactory(PersonQueue.class);

5    protected PersonQueue boys = queueFactory.create();
     protected PersonQueue girls = queueFactory.create();
7
     public void addBoy(Person boy) {
9      while (true) {
         DSTM2Thread.beginTransaction();
11       try {
           PersonQueueMethods.insert(boys, boy);
13       }
         catch (AbortedException e) {
15         // Conflict detected before committing
         }
17       if (DSTM2Thread.commitTransaction()) {
           break;
19       }
         // Conflict detected during committing
21     }
     }
23 }
```

Figure 3.1: Implementation of BallRoom using DSTM2

when a call to method commitTransaction() returns false (line 17). In either case, the control flow reaches the end of the loop and repeats. The body of method addGirls(...) (not shown) is similar to this; however, the pair() method cannot be implemented using DSTM2 due to the need for condition synchronization.

In practice, transactions suffer from several critical deficiencies that prevent them from being used in developing general purpose applications. First, although transactions appear to be more concurrent, they can cause serious performance issues when the systems are highly contentious and when conflicts are frequent [9]. In addition, repeatedly rolling back transactions is waste of electrical energy [45]. Second, transactions typically cannot im-

plement condition synchronization. Although researchers have proposed advanced features, such as conditional waiting [27], to ameliorate this situation, these features are still immature and are not widely adopted in existing transaction-based frameworks. Finally, transactions cannot be used in domains where operations on shared objects cannot be rolled back. An example of such domains is the IPT domain, where shared session objects are typically used to transmit messages over the network. What is more, shared objects can grow much too large to clone. Although the need to clone large objects can be ameliorated using transaction synchronizers, which permit multiple transactions to operate on the same objects [50], the runtime overhead of resolving conflicts can still be way too much.

3.1.3 Events

While transactions allow a programmer to emulate the effects of mutual exclusion without explicitly manipulating locks, events allow a programmer to signal threads using an API that is more intuitive and less error-prone than directly using wait() and notify() [57]. Intuitively, an event is an object, whose status can be changed and obtained by calling the methods on that object. The event reports its status change to an event manager, which monitors the status of all the events and notifies a thread when all the events that the thread subscribes to have occurred. Event subscription can be blocking or nonblocking. In the former case, a thread waits until it receives a notification from the event manager, similar to the situation when a thread blocks on a lock. In the latter case, a thread passes a callback object to the event manager and immediately resumes execution after subscribing to the events [30, 66]. When these events have all occurred, the event manager executes that callback object on another thread. Because this latter thread may access the same shared objects as the former thread (that subscribes to the events, these threads must use locks or transactions to enforce mutual exclusion). Blocking-based subscription is widely used in mainstream programming languages, such as Java and C#; whereas nonblocking-based subscription is more popular

in emerging languages, such as Erlang [3] and Scala [23].

Microsoft's .NET framework provides a library of classes that implement events using blocking-based subscriptions. One of these classes is AutoResetEvent, which models simple recurrent events. An instance of this class keeps track of the number of outstanding occurrences of an event and changes this number as threads produce or consume this event (similar to a semaphore). Another class is WaitHandle, which consists of numerous static methods for subscribing or manipulating events. Figure 3.2 illustrates an implementation of BallRoom using these classes. Specifically, the programmer declares two events, hasBoys (line 3) and hasGirls (line 6), to keep track of, respectively, the number of boys and the number of girls who have arrived but have not yet been paired. These numbers increment as threads set these events in addBoy(...) (line 20) and addGirl(...) (not shown). Method pair() subscribes to both events as the first thing it does (line 24). Because this subscription is blocking, a thread that executes this method will wait until both events have occurred, in which case the call to WaitAll(hasBoysAndGirls) returns. This call also consumes one occurrence of each event, decrementing the number of outstanding boys and the number of outstanding girls. Finally, the lock statements (lines 17 and 25), which are similar to the synchronized statements in Java, are necessary to prevent threads from concurrently accessing the queues.

As high-level abstractions, events can facilitate creation of more readable and reliable implementations of condition synchronization than Java synchronization primitives. However, it can be cumbersome or unintuitive to implement mutual exclusion using events. For example, in order to use events to emulate the mutual exclusion in Figure 3.2 (lines 17–19 and 25–28), the programmer would need to declare another event areQueuesFree and set the initial status of this event to be "signaled" (i.e., by passing value true to AutoResetEvent's constructor). Then, instead of using the lock keyword to create a synchronized block, the programmer would need to wait for that event on entry of that block and would need to

```
1  public class BallRoom {
     LinkedList<Person> boys = new LinkedList<Person>();
3    AutoResetEvent hasBoys = new AutoResetEvent(false);

5    LinkedList<Person> girls = new LinkedList<Person>();
     AutoResetEvent hasGirls = new AutoResetEvent(false);
7
     AutoResetEvent hasBoysAndGirls[] = new AutoResetEvent[2];
9    Object queueLock = new Object();

11   public BallRoom() {
       hasBoysAndGirls[0] = hasBoys;
13     hasBoysAndGirls[1] = hasGirls;
     }
15
     public void addBoy(Person boy) {
17     lock (queueLock) {
         boys.AddLast(boy);
19     }
       hasBoys.Set();
21   }

23   public void pair() {
       WaitHandle.WaitAll(hasBoysAndGirls);
25     lock (queueLock) {
         boys.RemoveFirst();
27       girls.RemoveFirst();
       }
29   }
   }
```

Figure 3.2: Implementation of BallRoom using events in .NET

set that event on exit of that block. Although such emulation is not elegant, it is at least possible to do so. For the systems that use nonblocking-based subscription, either locks or transactions are almost always needed to achieve the effects of mutual exclusion.

3.2 Automatic programming of synchronization

No matter which synchronization primitives a programmer chooses to use, if the programmer manually implements the synchronization logic and mixes the synchronization code with the functional code, the synchronization problems described in Chapter 2 still exist. To address these problems, recent research has mainly focused on separating the synchronization logic from a program's functional logic. Such separation can facilitate automatic programming of synchronization, thereby improving a program's modularity and maintainability. This section describes approaches to automatically generating the synchronization code and discusses how these approaches influenced our proposed solution.

3.2.1 Conditional critical regions

Hoare proposed conditional critical regions to unify mutual exclusion with condition synchronization [35]. Intuitively, a conditional critical region consists of an object that will be used when a thread executes that region and a boolean expression, called a guard, that queries the state of that object and that must be true before the thread executes that region. Briefly, A thread attempting to enter a conditional critical region must lock that object before evaluating that guard. Unless that thread successfully locks that object and that guard is true, the thread will unlock that object, wait for a certain amount of time, and retry. Conditional critical regions are not implemented in Java. Some early programming languages, such as C, implement conditional critical regions as a non-standard language extension [44]. Using this extension, conditional critical regions in a program will be translated into lock-based

code by a pre-processor.

Our contract-based synchronization approach is inspired by conditional critical regions. It treats a method as a conditional critical region and allows a programmer to declaratively specify the objects and the guards associated with that method. Different from the traditional concept, which associates a conditional critical region with only one object and one guard, our approach allows multiple shared objects and multiple guards to be associated with a method. Such flexibility helps a programmer write more reliable and readable code. First, a programmer can write a method that accesses multiple shared objects without nesting synchronized statements. Nesting synchronized statements can not only cause deadlocks but also make the code difficult to read. Second, because a programmer can specify multiple guards for a method, she can transform a complex guard into a conjunction of simpler ones and reuse these simpler ones for other methods.

3.2.2 Java with annotated concurrency (JAC)

The JAC approach [28] facilitates the programming of monitor classes. It allows a programmer to place special Java comments above a sequential class (i.e., not synchronized) or above the methods of that class to declaratively specify the synchronization requirements she expects to achieve. Example synchronization requirements that can be specified include which methods can be executed concurrently and which guards must be true before executing a method. JAC uses a custom Java pre-processor to automatically translate such special comments into implementation code and automatically embeds this code in the original class.

Figure 3.3 illustrates an implementation of BallRoom using the JAC approach. This implementation contains three JAC comments (lines 5, 10, and 15). The top two comments signify that methods addBoy(...) and addGirl(...) are compatible to each other—in other words, they can be executed concurrently without introducing data races. The bottom comment signifies that the boolean expression after @when must be true in order to execute

```
public class BallRoom {
  protected Queue<Person> boys = new LinkedList<Person>();
  protected Queue<Person> girls = new LinkedList<Person>();

  /** @compatible addGirl(Person) */
  public void addBoy(Person boy) {
    boys.add(boy);
  }

  /** @compatible addBoy(Person) */
  public void addGirl(Person girl) {
    girls.add(girl);
  }

  /** @when boys.size()>0 && girls.size()>0 */
  public void pair() {
    boys.remove();
    girls.remove();
  }
}
```

Figure 3.3: Implementation of BallRoom using JAC

method pair(). If this boolean expression is false when a thread invokes this method, the thread will wait and retry. Finally, because the comment above method pair() does not contain @compatible, this method cannot be executed concurrently with any other method in the class.

Our contract-based synchronization approach is similar in spirit to the JAC approach. The main difference is that our approach allows a programmer to specify the set of shared resources that a method needs to use, rather than the list of other methods that can be executed concurrently. In fact, whether two methods can be executed concurrently purely depends on whether these methods use the same resource. However, specifying the set of shared resources is more direct and more local. Being more direct allows a programmer to think more about the synchronization requirements instead of the implementation. Being

more local makes the code easier to extend and more manageable. For example, when a programmer extends a class with a new method, using our approach, she would need to specify only the resources used by that method in the contract. Using JAC, by contrast, the programmer would have to read all the existing methods in that class and find out those that can be concurrently executed with this new method. This discovery task can be very inefficient and error-prone, especially when the class consists of a large number of methods.

Additionally, our approach leverages Java annotations, rather than Java comments, to embed synchronization contracts. Because Java annotations are structured and typed, using them allows us to design a richer and safer contract language than using comments. Also, because Java annotations are standard, using them makes our approach easier to distribute and to be adopted by the practitioners.

3.2.3 Concurrency controllers

Concurrency controllers is a design pattern proposed by Betin-Can et. al. to facilitate separation of the synchronization code from the functional code [8]. Specifically, rather than directly mixing the synchronization code with the functional code, a programmer encapsulates the synchronization code in a concurrency controller class and invokes the methods of this class in the functional code. The programmer is required to write the concurrency controller class such that a call to a method of this class should return only when it is safe for the calling thread to proceed. Intuitively, a concurrency controller is analogous to a centralized authority, who grants execution permissions to threads, which are obligated to notify the controller in advance of what they intend to do.

Figure 3.4 illustrates an implementation of BallRoom using the concurrency controllers design pattern. This implementation creates an instance of the controller class (lines 2–3) and tells this instance when a thread enters a method (lines 9, 15, and 21) and when a thread exits the method (lines 11, 17, and 24). On entry to a method, this controller instance blocks

```
public class BallRoom {
  protected BRControllerInterface controller =
    new BRControllerUsingResourceNumbering();

  protected Queue<Person> boys = new LinkedList<Person>();
  protected Queue<Person> girls = new LinkedList<Person>();

  public void addBoy(Person boy) {
    controller.addBoy_enter();
    boys.add(boy);
    controller.addBoy_exit();
  }

  public void addGirl(Person girl) {
    controller.addGirl_enter();
    girls.add(girl);
    controller.addGirl_exit();
  }

  public void pair() {
    controller.pair_enter();
    boys.remove();
    girls.remove();
    controller.pair_exit();
  }
}
```

Figure 3.4: Implementation of BallRoom using a concurrency controller

a thread until all the synchronization requirements, which are hard-coded in the controller class, of that method are met. On exit from a method, this controller instance releases resources acquired on the entry.

The advantage of using concurrency controllers arises from the fact that the synchronization code is loosely coupled. Therefore, a program can easily change its synchronization protocol by changing its concurrency controller class. For example, the programmer defines BRControllerInterface (line 2) as the interface for the concurrency controllers that can be

used in the BallRoom class. The programmer then implements a concrete controller class BRControllerUsingResourceNumbering using the resource numbering synchronization protocol. To use a different synchronization protocol, the only change that the programmer needs to make is to instantiate a new controller class on line 2.

In spite of these advantage, manually implementing a concurrency controller class can be very tedious and error-prone. Our contract-based synchronization approach attempts to address this issue by automatically generating a concurrency controller class from declarative synchronization contracts. In fact, the idea of generating synchronization code from high-level specifications is not new; for example, Dwyer has proposed an approach to generate synchronization code from region invariants [12]. However, the existing approaches based on this idea all generate tightly coupled synchronization code, which makes it difficult to change synchronization protocols. By contrast, the code generated by our approach essentially instantiates the concurrency controllers design pattern, similar to the code in Figure 3.4. Furthermore, we noticed that most concurrency controller classes have common parts in their implementations, which are typically used to handle mutual exclusion and condition synchronization. Our approach further separates and encapsulates these common parts into highly reusable components, called negotiators. Such separation can potentially improve a program's reliability and efficiency, because the implementation of negotiators can be provided by synchronization experts and third-party vendors.

3.2.4 Aspect-oriented programming (AOP)

Separation of concerns has been a general principle in software engineering for handling a program's complexity [54]. AOP is probably the first programming paradigm that embodies this principle on the programming-level [41]. Specifically, AOP allows a programmer to implement a program's crosscutting concerns separately and weave the implementation of these concerns into the functional code using a weaving tool. Here, a crosscutting concern is

```
aspect AddConcurrencyControllerToBallRoom {
  pointcut add_boy_method(Person p): this(BallRoom) &&
      args(p) && execution(void addBoy(Person));

  void around(Person boy): add_boy_method(boy) {
    controller.addBoy_enter();
    proceed(boy);
    controller.addBoy_exit();
  }
  ...
}
```

Figure 3.5: Implementation of a concurrency controller aspect using AspectJ

implemented as an aspect, which consists of pointcuts and advices. A pointcut defines the locations in the functional code where additional code will be joined. An advice specifies the new code to be woven before, after, or around a pointcut.

AspectJ is an aspect-oriented extension to Java [42]. Figure 3.5 illustrates an example of an aspect implemented using AspectJ. This aspect aims to weave calls to the concurrency controller into BallRoom. Pointcut add_boy_method (lines 2–3) locates the body of addBoy(...) (indicated by predicate execution(...)) in class BallRoom (indicated by predicate this(...)). Advice around (lines 5–9) tells the AOP weaving tool to join code (lines 6 and 8) in the beginning and in the end of addBoy(...), which is referred to by proceed(...). Using this advice and the functional code shown in Figure 2.2, the AOP weaving tool will modify BallRoom such that the modified class is equivalent to that in Figure 3.4.[4] Although it appears that using AOP is like using a sledgehammer to crack a nut in this example, AOP can be extremely helpful when advices need to be applied on large numbers of locations in the source code.

Some concerns that AOP is good at handling include logging, tracking, and caching [48,

[4]The modified class will be much longer because the AOP weaving tool also generates some wrapping methods and utility functions in the class.

56]. The synchronization concerns, however, have proven to be expressed using AOP [43, 4]. The main problem of AOP is that the pointcut model, which is based on the code's syntactic appearance, is overly simplistic. The synchronization code typically depends on the functional code and, therefore, must be "blended", rather than simply "inserted". As a consequence, most approaches that attempt to use AOP in addressing synchronization concerns are very limited. For example, Concurrency Aspect Library (CAL) is a library of reusable aspects aiming to parallelize a method call in a sequential code [71]. Parallelization is achieved by wrapping a method call into a Runnable class, which is then executed by a thread. It is feasible to use AOP in wrapping method calls, because the code that CAL inserts has a fixed pattern, which does not depend on the target method calls. As another example, FlexSync is an aspect-oriented approach to adding mutual exclusion to existing methods [70]. A programmer can choose whether the mutual exclusion will be achieved by locks or transactions. In the former case, FlexSync inserts a synchronized keyword for a method; in the latter case, FlexSync inserts transactional calls—e.g., beginTransaction() and commitTransaction()—around a method. However, FlexSync cannot add condition synchronization to existing methods, as the code implementing condition synchronization is highly sensitive to the details of the target method.

Our contract-based synchronization approach differs from AOP in that our approach not only "blends" the synchronization code into the functional code but also generates the synchronization code. AOP, by contrast, focuses only on code weaving; the synchronization code still needs to be written by the programmer. For example, the aspect in Figure 3.5 does not generate the concurrency controller class. Moreover, our approach can handle both mutual exclusion and condition synchronization.

This chapter described recent approaches to automatic programming of the synchronization concerns and the lessons we learned from these approaches. The next chapter describes our contract-based synchronization approach. Instead of simply repeating our previous work

on Szumo, which is the foundation of our approach, the next chapter focuses on their differences.

Chapter 4

Contract-based Synchronization

Our contract-based synchronization approach evolves from our previous work on Szumo [6, 63, 7]. Similar to Szumo, our approach also leverages declarative synchronization contracts to separate the synchronization logic; however, our approach makes the following changes.

- Our approach associates synchronization contracts with methods.
- Our approach does not support vertical composition of synchronization constraints.
- Our approach generates loosely coupled synchronization code using the concurrency controllers design pattern.

This chapter describes these changes and discusses the rationales behind them.

4.1 Associating contracts with methods

In Szumo, a synchronization contract is associated with a class. Intuitively, a synchronization contract defines a set of abstract synchronization states and declares the shared objects that an instance of that class assumes it may access in each of the states. A programmer, who implements the class, writes code that maintains a set of condition variables to encode and

Table 4.1: Synchronization contract for BallRoom in Szumo

Sync. state	Condition	Shared objects	Guard
adding_boy	isAddingBoy	{boys}	true
adding_girl	isAddingGirl	{girls}	true
pairing	isPairing	{boys, girls}	boys.size()¿0 and girls.size()¿0

reflect the current synchronization state of the execution. In response to changes of these condition variables, a runtime system schedules some threads and suspends others to enforce mutual exclusion and condition synchronization based on the synchronization contracts.

For example, Table 4.1 illustrates a synchronization contract that a programmer might come up with for class BallRoom in Szumo.[1] This contract defines three synchronization states (1st column). For each state, the contract declares a boolean expression over condition variables (2nd column),[2] a collection of shared objects (3rd column), and a guard (4th column). The condition is used to indicate whether an instance of this class is in a synchronization state (i.e., if the condition is true) or not (i.e., if the condition is false). It is the programmer's responsibility to update the values of condition variables so that the conditions encode the intended semantics: in this example, a condition variable in an instance of the ballroom class should be true only if a thread is executing the designated method on that instance (e.g., an ballroom's isPairing field is true only if some thread is executing its pair method). The collection of shared objects signifies the resources that are accessed in that state. The guard signifies the boolean condition that must be true to enter that state. The guard true is always true, and so essentially means that there is no guard condition.

[1]The syntax used in Szumo is more constraint-like and does not name the states. For example the first and last rows are expressed, respectively:
isAddingBoy =¿ boys
isPairing =¿ (boys and girls) when (boys.size()¿0 and girls.size()¿0).

[2]In this example, condition variables are in one-to-one correspondence with states because this example cannot be done with fewer condition variables. In general, there may be fewer condition variables than concurrency states and the second column of the table (left side of the Szumo constraint) may be a boolean expression.

```
1  public class BallRoom {
     protected Queue<Person> boys = new LinkedList<Person>();
3    protected Queue<Person> girls = new LinkedList<Person>();

5    protected boolean isAddingBoy = false;
     protected boolean isAddingGirl = false;
7    protected boolean isPairing = false;

9    public void addBoy(Person boy) {
       isAddingBoy = true;
11     Szumo.negotiate(this);
       boys.add(boy);
13     isAddingBoy = false;
       Szumo.negotiate(this);
15   }

17   public void addGirl(Person girl) { ... }

19   public void pair() { ... }

21   // Below code is generated from contract
     public Set<Object> getSharedResources() {
23     Set<Object> resources = new HashSet<Object>();
       if (isAddingBoy || isPairing) resources.add(boys);
25     if (isAddingGirl || isPairing) resources.add(girls);
       return resources;
27   }

29   public boolean areGuardsSatisfied() {
       boolean result = true;
31     if (isPairing)
         result &= boys.size() > 0 && girls.size() > 0;
33     return result;
     }
35 }
```

Figure 4.1: Implementation of BallRoom using Szumo

Table 4.2: Synchronization contract for BallRoom in our approach

Method	Shared objects	Guard
addBoy(Person)	boys	true
addGirl(Person)	girls	true
pair()	boys, girls	boys.size()¿0 and girls.size()¿0

Figure 4.1 illustrates how this contract can be implemented (lines 22–34) in the BallRoom class. This implementation of the contract follows the conventions in our prior SzumoC++ framework—a reference implementation of Szumo for C++—and can be automatically generated from the contract. Specifically, method getSharedResources() returns a set of objects that are required by any of the current synchronization states based on the values of the condition variables. Method areGuardsSatisfied() returns true if all the guards for the current synchronization state are true or returns false otherwise. These methods are called by the Szumo runtime whenever a BallRoom instance changes the values of the condition variables (lines 10 and 13) and notifies the Szumo runtime to renegotiate the contract (lines 11 and 14). In negotiating the contract, the Szumo runtime will block the execution of that instance if any of the objects that getSharedResources() returns is not available or if areGuardsSatisfied() returns false.

In our approach, we assume that a method body always enters a synchronization state as the first thing it does and leaves the state as the last thing it does. This assumption allows us to remove synchronization states from synchronization contracts and associate synchronization contracts with method bodies. As a consequence, a programmer specifies shared objects and guards for a method body, rather than for a set of synchronization states. Table 4.2 illustrates the synchronization contracts for the methods of the BallRoom class in our approach. These contracts express essentially the same information as the example Szumo contracts (Table 4.1); however, because the condition is implicitly "executing the method body," there is no need to specify conditions. As a result, a programmer does not

need to embed code that manipulates condition variables or notifies the Szumo runtime in the functional code. Instead, in our approach, a compiler plugin wraps the method body in order to automatically negotiate the method's contract when a thread invokes the method and automatically notify other threads when the method returns. As a result, the Java code that a programmer writes using our approach will contain only the functional code. For example, for the BallRoom class, a programmer writes identical functional code to that shown in Figure 2.2. To this functional code, the programmer then adds synchronization annotations (Ch. 5).

Our decision to associate synchronization contracts with method bodies is influenced by the Single Responsibility Principle (SRP) [52], a widely adopted design principle in software engineering. Methods that are written following SRP tend to be shorter and more consistent—a common criteria for measuring the quality of methods [19]—than those that have multiple responsibilities. Assuming that a method is responsible for performing some functional logic, SRP suggests that it should not also be responsible for performing synchronization midway during execution. In other words, the synchronization requirements of a well-designed method typically should not change in the middle of executing the method body. The typical places where synchronization should occur are at method boundaries—i.e., when a method is invoked and when a method returns.

4.2 No support of vertical composition of synchronization constraints

One of the benefits that Szumo promises is the ability to compose synchronization constraints. Here, a synchronization constraint with condition (second column of Fig. 4.1), C, specifies a set of shared objects that will be needed when in a synchronization state in which C is true and a guard condition that must be true in order to enter such a state. Szumo provides two types of composition—horizontal composition and vertical composition.

Horizontal composition occurs when the conditions in multiple constraints are true in the same synchronization state. In this case, the Szumo runtime combines all the shared objects required by any of these conditions (thus horizontally composing the synchronization constraints associated with these condition variables). Our approach supports a similar type of composition at the level of guard conditions.

Vertical composition occurs when the Szumo runtime composes the synchronization constraints of an object's contract with the synchronization constraints of another object's contract. For example, when negotiating the synchronization contract for an object, u, the Szumo runtime migrates into u's realm not only the shared objects that u needs to use in u's synchronization states (i.e., by calling u.getSharedResources()) but also the shared objects that each of the migrated objects, v, needs to use in v's synchronization states (i.e., by calling v.getSharedResources()). Therefore, this process of migrating objects can be seen as composing u's synchronization constraints with v's synchronization constraints. Because v is needed by u, such composition is considered as vertical. Vertical composition is completed when it reaches a fixed point—i.e., when u's realm subsumes all the objects needed by any object in the realm.

Our approach does not yet support vertical composition for two reasons. First, omitting vertical composition allowed us to more quickly produce a reference implementation in order to experiment with the overall approach. Second, it is not clear whether vertical composition is really needed in practice. In the concurrent programs we investigated in our case studies, we have not seen any cases that warranted adding vertical composition. However, these programs may not be fully representative of concurrent programs in practice. We believe that a useful definition of vertical composition would necessitate a more expressive annotation language.

Both Szumo and our approach facilitate deadlock prevention by reducing the need for a thread to acquire resources incrementally. But neither approach precludes the possibility

of deadlock. In our approach, for example, a thread may call a synchronization method, m_1, within the body of another synchronization method, m_0; and this latter method may similarly call the former within its body. If two threads, t_0 and t_1, invoke, respectively, m_0 and m_1 on the same object, o, and if evaluation of the contracts for m_0 and m_1 on o indicates that the two methods need non-empty, but disjoint, sets of resources, then deadlock can easily occur. It is conceivable that vertical composition might reduce the need to nest calls to synchronization methods in this fashion. Such nesting is akin to nesting monitor calls, which many software engineering experts consider a bad software design [49, 51]. This issues deserve additional attention.

4.3 Instantiating concurrency controllers

In Szumo, the synchronization details that are used for negotiating contracts implement a complex synchronization protocol based on two-phase (claim and commit) locking. Although this protocol admits more concurrency than simpler protocols and is able to avoid a large class of deadlock cases, it introduces significantly more runtime overhead in cases where a simpler protocol suffices [38]. Our previous experiments showed that a web server based on the same design as the Apache Web Sever, but written in Szumo/Eiffel, was not nearly as efficient in its use of locks as a commercial Apache server. Unfortunately, our initial implementation of Szumo does not allow us to replace this complex synchronization protocol with another one, simply because this protocol has been hard-coded into the Szumo runtime. To address this issue, therefore, our approach leverages the concurrency controllers design pattern (Section 3.2.3) to decouple the generated code from synchronization details. Specifically, the code generated by our approach instantiates this pattern, rather than implementing any specific synchronization protocol. The synchronization protocols are encapsulated in separate and reusable negotiator components, which are provided by the synchronization experts

or third-party vendors. Such separation allows a programmer to choose a synchronization protocol based on the usage scenarios that she expects and to dynamically replace an existing protocol without modifying the functional code. Researchers have found such flexibility to be very useful in developing and maintaining programs [71, 70, 38].

This chapter contrasts our contract-based synchronization approach with Szumo. The next two chapters describe how we integrated our approach with a mainstream programming language and an existing middleware platform. Specifically, Chapter 5 describes how we integrated synchronization contracts with Java and how we generated synchronization-aware classes from the contracts and the functional code that a programmer writes. Chapter 6 describes how we integrated synchronization contracts with standard SIP servlets containers in the IPT domain and how we dynamically enforced contracts in a synchronization middleware at runtime.

Chapter 5

Generative Approach

Our generative approach to integrating synchronization contracts with multi-threaded Java programs leverages the standard Java annotation mechanism to specify synchronization contracts (Fig 5.1). Briefly, an application programmer implements the functional logic as a set of Java classes and embeds synchronization annotations into each class whose instances may be accessed by multiple threads. In this chapter, we refer to this program as the application program. The synchronization annotations specify the conditions under which it is safe[1] to execute a method on such a shared object and the fields of the object that the method manipulates. When it encounters a class containing a synchronization annotation, the Java compiler automatically invokes a synchronization annotation processor (SAP), which generates a set of derived classes from the annotated Java program. The derived classes invoke operations of a synchronization service to automatically synchronize threads as they invoke methods on shared objects. We therefore refer to the program obtained by combining the application program and the derived classes generated from it as the synchronization-enhanced program. Because the synchronization service is not application specific, it can be supplied by a third party as a Java library. After loading a synchronization service, the byte code

[1]The notion of "safety" referred to here is a functional one, which is typically formalized as a class invariant [55].

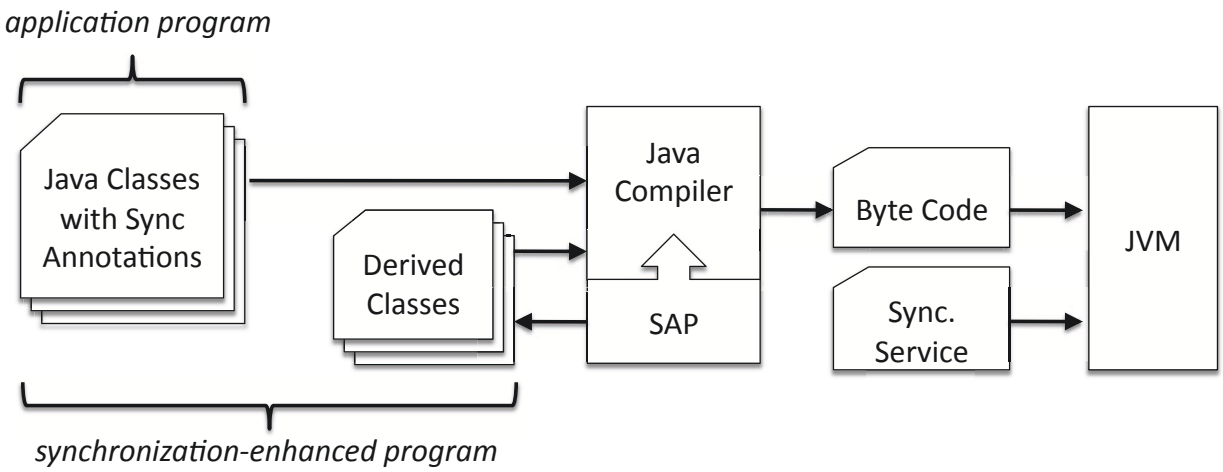


Figure 5.1: A Generative Approach Using Synchronization Annotations in Java

produced from compiling the synchronization-enhanced program can be executed within any JVM.

This chapter describes the details of this generative approach. We first describe the form and meaning of our synchronization annotations for Java (Sec. 5.1). Because the derived classes depend on a synchronization service library, we describe the services provided by this library (Sec. 5.2) next, and then the derived classes and the process by which the SAP generates them (Sec. 5.3). Next, we report results of a study in which we programmed solutions for a variety of problems that we found in surveying the literature on concurrent programming (Sec. 5.4). The chapter wraps up with some observations about the limitations of the generative approach (Sec. 5.5).

5.1 Synchronization Annotations for Java

The UML class diagram [60] in Figure 5.2 provides a meta model for programs containing synchronization annotations. A class (i.e., instance of meta-type Class) contains methods and fields (instances of, resp., meta-types Method and Field), where a method encapsulates

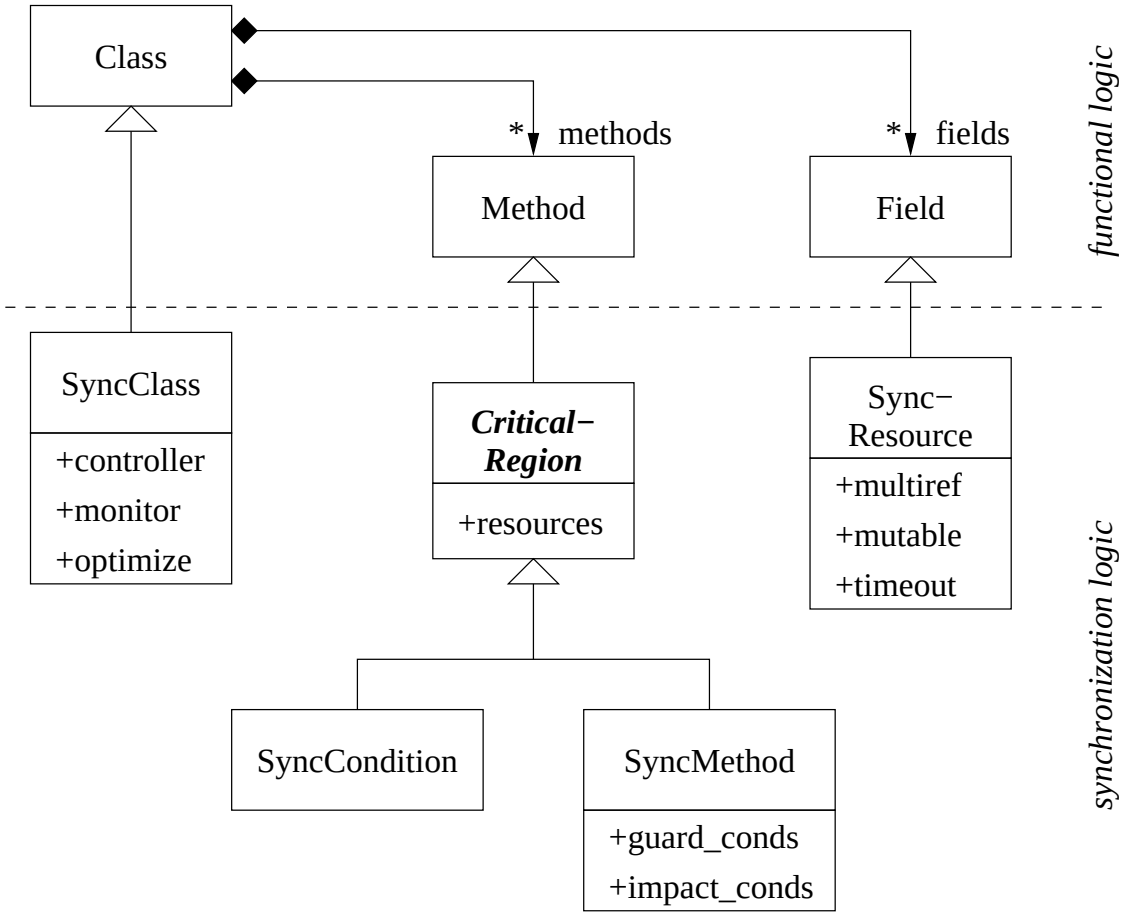


Figure 5.2: Meta Model for Programs Containing Synchronization Annotations

code to be executed on an instance of the class and fields refer to the data objects that, taken together, comprise the instance's state. Each of these plain Java constructs—classes, methods, and fields—may contain a synchronization annotation. For brevity, we refer to an annotated class as a sync class, an annotated method as a sync method, and an annotated field as a sync resource (the latter because the field references a shared resource, i.e., an object that can be accessed by multiple threads).

Intuitively, the plain Java constructs express the application's functional logic, while the synchronization annotations express information about when and how threads need to manipulate shared resources. Because a sync class implements only functional logic, its in-

stances are not thread safe. To prevent instantiation, we therefore require a sync class to be abstract. We also require that each sync class declares at least one factory method [21] of the form:

```
static public SyncClassName instance(...) {
  return new SyncClassControllerName(...);
}
```

where SyncClassName denotes the name of the sync class and SyncClassControllerName denotes the value of the controller property in the sync-class annotation (discussed later in this subsection). To create an instance of the sync class, the application program must use one of these factory methods. Our reference SAP generates the class named SyncClassControllerName so that it subclasses the sync class, wrapping the sync class in a sync controller class. Thus, transparent to the application programmer, each instance of a sync class is a sync controller of type SyncClassControllerName. The sync class BallRoom in Figure 5.3 illustrates these conventions. An abstract class (line 2), it provides a factory method (lines 9–11), which is the only means by which client classes (e.g., classes whose instances represent boys, girls and teachers) can obtain a BallRoom object. Transparent to the application programmer, the instance returned by this factory method is a sync controller of type BallRoomSync.

A synchronization annotation consists of an annotation name, which starts with an "@" symbol, and a set of properties. A property binds a property name to a property value. In a program, a synchronization annotation is written in the form

$$\text{@ann-name}(\text{prop-name}_1 = v_1, \ldots, \text{prop-name}_n = v_n)$$

where @ann-name denotes the annotation name, and prop-name$_i$ and v_i denote, respectively, a property name and its value, for $1 \leq i \leq n$ and $n \geq 0$. The annotation name determines

```
1  @SyncClass(controller = "BallRoomSync")
   abstract public class BallRoom {
3    @SyncResource
     protected Queue<Person> boys = new LinkedList<Person>();
5
     @SyncResource
7    protected Queue<Person> girls = new LinkedList<Person>();

9    static public BallRoom instance() {
       return new BallRoomSync();
11   }

13   @SyncCondition(resources = {"boys"})
     public boolean hasBoys() { return boys.size() > 0; }
15
     @SyncCondition(resources = {"girls"})
17   public boolean hasGirls() { return girls.size() > 0; }

19   @SyncMethod(resources = {"boys", "girls"},
                 guard_conditions = {"hasBoys", "hasGirls"})
21   public void pair() {
       boys.remove();
23     girls.remove();
     }
25
     @SyncMethod(resources = {"boys"},
27               impact_conditions = {"hasBoys"})
     public void addBoy(Person boy) {
29     boys.add(boy);
     }
31
     @SyncMethod(resources = {"girls"},
33               impact_conditions = {"hasGirls"})
     public void addGirl(Person girl) {
35     girls.add(girl);
     }
37 }
```

Figure 5.3: Implementation of BallRoom using synchronization annotations

Table 5.1: Properties of the synchronization annotations

Annotation Name	Property Name	Property Type	Default Value
@SyncClass	controller	string	N.A.
@SyncClass	optimize	boolean	FALSE
@SyncClass	monitor	boolean	FALSE
@SyncResource	multiref	boolean	FALSE
@SyncResource	mutable	boolean	TRUE
@SyncCondition, @SyncMethod	resources	set of strings	∅
@SyncMethod	guard_conditions	set of strings	∅
@SyncMethod	impact_conditions	set of strings	∅
@SyncMethod	timeout	long (milliseconds)	0

the type of annotation, indicating where the annotation may be used, the set of properties it may include, and its meaning. Property names are typed and most have default values; Table 5.1 shows the types of all properties and their defaults.[2]

The four types of synchronization annotations and their properties are:

@SyncClass: Signifies that an instance of the class may be accessed concurrently by multiple threads (a sync-class annotation). A sync class may (optionally) be nested within another sync class. It must be abstract to prevent instantiation and it cannot be final because our SAP specializes it to produce the sync-controller class. Its properties specify:

controller: the name to be used for the sync-controller class. This name must be different from the name of any other class in the application. (A required value, it has no default.)

monitor: whether instances of the class are monitors [33] or not (the default). A sync controller for which monitor is true will serialize executions of its methods by multiple threads.

[2]In the future, we plan to change the defaults for optimize, mutable, and multiref.

optimize: whether to perform an optimization that reduces memory usage or not (the default).

For example, the sync-class annotation in Figure 5.3 specifies a name for the sync-controller class only (line 1); by default, therefore, a BallRoom object is not a monitor and its implementation is not optimized.

@SyncResource: Signifies that the field references an object that can be read or updated by multiple threads (a sync-resource annotation). Each sync resource must be contained in the scope of some sync class. To ensure that a sync resource is visible within classes generated by our SAP, but that it is not directly manipulated by application code, we do not permit it to be private or public. The two properties for sync resources specify:

mutable: whether the object referenced by the field is mutable (the default) or not.

multiref: whether the field contains the sole reference to this object (the default), or if multiple references to the object may exist.

If mutable is false, the resource cannot be wrapped as efficiently as if it is true. We do not permit mutable to be false and multiref to be true at the same time (for reasons explained in Section 5.3.1). Our example illustrates the common case. The two sync-resource annotations (Fig. 5.3, lines 3 and 6) indicate that the boys and girls fields of a BallRoom object each references a distinct mutable (by default) queue of persons. Moreover, multiple threads may read and update either or both of these queue(s); but only by invoking methods on the given BallRoom object (also, by default).

@SyncCondition: Annotates a method that is used by another method as a boolean guard (a sync-condition annotation). A sync condition must be in the scope of some sync class. Acting as a guard, it queries the state of a sync controller (the receiver object) to determine if some other method can be safely executed; it must therefore return a

boolean value. Additionally, it must not produce any side effects, as the sync controller may need to invoke it an indeterminate number of times.[3] A sync condition must be public, as it needs to be visible in the synchronization controller and needs to be accessible by the synchronization service (see Section 5.2). However, it should not be invoked directly by an application program.[4] A sync condition has only one property, which specifies:

resources: the names of the sync resources that the method reads. (A set of strings, it is empty by default.)

The BallRoom class in Figure 5.3 contains two sync-condition annotations. The first sync condition (lines 13–14) checks that the queue referenced by the boys field of a BallRoom object is non-empty; while the second (lines 16–17) performs the same check, but on the queue referenced by the girls field. In each case, the resources value indicates the queue that must be read to evaluate the condition.

@SyncMethod: Signifies that the method may read and update the state of the receiver sync-class instance (a sync-method annotation). A sync method must be in the scope of some sync class. It must not be private in order that client classes may invoke it, and it must not be final because the sync-controller class redefines it. The properties of a sync method specify:

resources: the names of any sync resources that the method reads and/or updates. (A set of strings, it is empty by default.)

guard_conditions: the names of any sync conditions that must be true for the sync method to safely execute. (A set of strings, it is empty by default.)

[3] One consequence of this latter requirement is that the method may not contain a throws clause.
[4] Not enforced by our current implementation.

impact_conditions: the names of any sync conditions whose values could become true as a result of evaluating the method body. (A set of strings, it is empty by default.)

timeout: either zero (the default) or the maximum number of milliseconds that a controller may delay execution of the sync-method's body before aborting a call.

The BalllRoom class declares three sync methods (Fig. 5.3). In the sync-method annotation for pair, the resources property (line 19) indicates that executing pair may modify the queues referenced by the receiver's boys and girls fields, and the guard_conditions property (line 20) indicates that the body of pair may be safely executed only if the queues referenced by the receiver object's fields are both non-empty (i.e., both sync conditions return true). Additionally, the default impact_conditions property indicates that, if one of the sync conditions is false, executing pair cannot cause it to become true, and the default timeout property indicates that a BallRoom object can delay a thread invoking pair for an indefinite amount of time. In contrast, the sync-method annotation for addBoy (lines 26–27) indicates that executing addBoy can modify the queue referenced by the receiver object's boys field, potentially causing the queue to become non-empty (i.e., causing hasBoys to become true).

The key idea behind synchronization annotations is as follows: When multiple threads invoke sync methods on the same sync controller, the controller uses the annotations to determine if any of the calls should be aborted and to schedule the executions of the associated method bodies for the calls it does not abort. We say a call on a sync method of a sync controller is enabled if the thread that executes the call has exclusive access to all of the resources it needs to evaluate the method's guard conditions and to execute the method's body, and if the method's guard conditions are all true. When the method is called, the controller should either delay execution of the method body until it succeeds in enabling the

call or, if the value of the method's timeout property is non-zero and the call is not enabled within the time specified, abort the call. Viewing the synchronization annotations as software contracts [55], we assume the synchronization annotations specify the resources that a thread needs to execute the method body and to evaluate the guard conditions. Specifically, it needs exclusive access to all resources indicated by the method's resource property and by the resource properties of all sync conditions named in the method's guard_conditions property. We refer to the set of resources that a thread has exclusive access to as the thread's holds set. If the call to the sync method is made while executing the body of another sync method, the thread's holds set may already contain some of the resources indicated by the synchronization annotations; but if any of the indicated resources are not contained in the thread's hold set when the call is made, the controller must acquire these resources before proceeding to execute the method body. Before returning control to the caller, if any other controllers are waiting for a sync condition to become true and that sync condition could become true by execution of the method body, the controller should notify these other controllers to re-check the condition. Again, viewing the synchronization annotations as software contracts, we assume the impact_conditions property of the sync method specifies all sync conditions that could be impacted. Also before returning, the controller should restore the thread's holds set to what it was when the method was called, a step referred to as contracting the holds set.

In summary, when a thread calls a sync method, m, on a sync controller, c, there are two possible outcomes, depending on how quickly the call becomes enabled and on whether m's timeout value, tm, is 0 or not.

1. If tm is greater than 0 and c does not succeed in enabling the thread within tm milliseconds, c aborts the call, returning to the caller without acquiring any new resources or executing m's body.

2. Otherwise, c enables the call and executes m's body. Then, prior to returning control to the caller, c contracts the thread's holds set and, if any other controllers are waiting for a sync condition that is named in m's impact_conditions property to become true, notifies these controllers.

The design for a sync controller is meant to guarantee that the synchronization-enhanced program generated from an application program is free from data races, while also allowing use of standard synchronization policies for avoiding deadlock and starvation. But for a model of software contracts to provide any guarantees, the program must satisfy its contract [55]. In explaining the key idea behind our synchronization annotations, we noted several assumptions about the relationship between an application program and its synchronization annotations. Thus, in the case of synchronization annotations for Java, the satisfaction relation must ensure, at a minimum, that these assumptions are warranted.

Formal definitions of Java synchronization annotations and of this satisfaction relation are beyond the scope of this thesis. Instead, we provide guidelines that an application programmer should follow to obtain the expected benefits from using our approach:

1. If multiple threads can invoke modifier methods on the same instance of a class, then the class should be a sync class (i.e., it should have a sync-class annotation; it should be abstract; and it should contain one or more factory methods of the required form).

2. If the monitor property of a sync class is false, then each non-final field of the sync class should be a sync resource (i.e., it should have a sync-resource annotation and it should not be public or private); moreover, if the value of mutable is true, then the field should refer to a mutable object, and if the value of multiref is false, then the field should contain the sole reference to this object.

3. Each public method of a sync class should be one of:

(a) a constructor

(b) a factory method of the required form

(c) a sync method (i.e., it should have a sync method annotation and it should be public).

4. Each name that appears in the guard_conditions value of a sync method should be the name of a sync condition (i.e., a protected, side-effect free, boolean-valued function with a sync-condition annotation) in the same sync class as the sync method.

5. Each object that is read in evaluating a sync condition should be referenced either by a sync resource that is named in the sync condition's resources property or by one of the sync condition's formal parameters.

6. Each object that is read or modified in executing a sync method should be referenced by a sync resource that is named in the sync method's resources property, by a sync resource that is named in the resources property of a sync condition that is named in the method's guard_conditions property, or by one of the sync method's formal parameters.

7. For each sync condition and sync method in the same class, if the sync condition can become true by executing the body of the sync method, then the sync condition's name should appear in the sync-method's impact_conditions property.

8. The application code should not directly invoke any sync condition.

9. The application code should not directly implement any thread synchronization (e.g., create or use mutex locks, use the Java synchronized statement, etc.).

Our reference SAP checks a number of these criteria (Sec. 5.3). Moreover, we believe that, for most "correct" application programs, those criteria that we do not currently check can

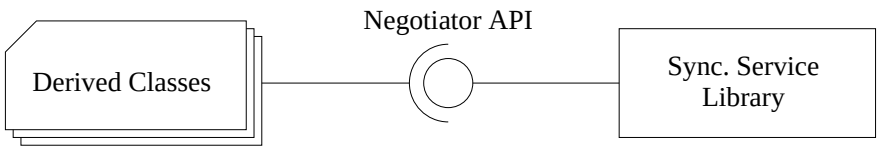


Figure 5.4: Relationship between a synchronization service and derived classes

be readily discharged using more sophisticated program analysis methods than the simple checks performed by our SAP. It is also possible that some of these criteria could be weakened without compromising safety in order to reduce the annotation burden placed on a programmer.

5.2 The Synchronization Service

A synchronization service provides basic synchronization functions, such as blocking a thread until some certain condition is met, etc. The derived classes that our SAP generates use such functions to automatically synchronize threads. Figure 5.4 illustrates the relationship between a synchronization service and the derived classes. Briefly, a synchronization service is supplied as a Java library, which typically implements a specific synchronization protocol using specific synchronization primitives. To abstract away such implementation details, the library exposes to the derived classes an implementation-neutral interface, called the Negotiator API. Using this API, a third-party software vendor may provide the synchronization service, and the derived classes are able to work with any synchronization service, regardless of the underlying synchronization protocol used and how it is implemented. A synchronization service library must maintain a unique negotiator for each thread in an application program, each of which must implement the Negotiator API. This section explains and motivates the negotiator-per-thread requirement (Section 5.2.1) and the Negotiator API (Section 5.2.2). It then describes a reference implementation that we developed (Section 5.2.3).

5.2.1 Negotiator-per-thread requirement

Intuitively, each thread in an application needs to own a unique negotiator. Briefly, a thread consults its negotiator for permission prior to executing the body of a sync method, and informs its negotiator for bookkeeping purposes after. When consulting its negotiator, a thread passes the negotiator a set of requirements, which encode the resources the thread needs in its holds set and the conditions that must be true in order to safely execute the method body. Upon being consulted, the negotiator blocks its owner until either it is able to satisfy these requirements or, if it is also passed a positive timeout value, a timer initialized to this value expires. In the former case, the negotiator may augment its owner's holds set with additional resources in order to satisfy some of the requirements. This potential side-effect of consulting a negotiator necessitates informing the negotiator after the thread finishes executing the method body. When a thread finishes executing the body of a sync method, it no longer requires exclusion on any resources acquired when it consulted the negotiator for permission to execute said body. Thus, informing a negotiator permits it to contract its owner's holds set.

A synchronization service library is responsible for managing negotiators and for supplying the negotiator owned by a thread when the thread requests it. The library must ensure that (1) different threads map to different negotiators and that (2) the negotiator that a thread maps to does not change over the thread's lifetime. Formally, this requirement can be stated as follows: a synchronization service must maintain an injective function, f, from the set of threads, T, into a set of negotiators, N. More specifically, for all t_0 and t_1 in T, if $t_0 \neq t_1$, then $f(t_0) \neq f(t_1)$. The reason we need an injective function relates to the usage scenario described in the previous paragraph: an opening operation (consulting the negotiator) is to be followed eventually by a closing operation (informing the negotiator) to allow maintenance of a thread's holds set. Consider what happens if a thread invokes a sync method, m_0, on one object and if, during execution of m_0's body, it then invokes

Table 5.2: Methods of SyncNegotiator

Method Name	Static	Return Type	Parameter Types
getNegotiator	Yes	SyncNegotiator	N.A.
trySatisfy	No	SyncNegotiatorResponse	SyncSentry[], long
contract	No	void	N.A.
notify	No	void	ConditionalSyncSentry

a sync method, m_1, on a different object. If both method bodies need exclusive access to the same resource, this resource will already be in the thread's hold set when it consults its negotiator for permission to execute m_1's body. By virtue of the fact that the opening and closing operations corresponding to m_0 and to m_1 are all invoked on the same negotiator—namely, the negotiator owned by the thread that invokes the sync methods—even though the methods operate on different objects, the negotiator can keep the history necessary to infer any additional resources the thread will need to execute m_1 and to match the next closing operation with the opening operation for m_1 so that it knows to release just those resources added to the thread's holds set by this opening operation. Such paired operations are very common in synchronizing threads, regardless of the underlying synchronization primitives.

5.2.2 Negotiator API

A negotiator is an instance of class SyncNegotiator, which defines a set of abstract methods to be implemented by a synchronization service library. Table 5.2 gives an overview of these methods. The following describes them in more detail:

getNegotiator(): A static method that returns the SyncNegotiator instance mapped to the calling thread. Calling this method is like computing the aforementioned function $f(t)$ with t being the thread that executes the call.

trySatisfy(reqs, tout): An instance method (non-static) that may potentially block the calling thread up to tout milliseconds while trying to satisfy the requirements represented

by reqs. If tout is 0, then it may block the calling thread for an indefinite amount of time. The first parameter, reqs, is an array of SyncSentry instances. A sync sentry (instance of type SyncSentry) responds to two queries: getResources(), which returns a set of sync resources; and evaluateCondition(), which returns a Boolean value. A negotiator has satisfied a sync sentry, s, when the holds set of its owner contains all of the resources in s.getResources() and s.evaluateCondition() returns true. A call to trySatisfy(reqs, tout) returns an instance of SyncNegotiatorResponse, indicating one of the following results.

- Positive: The requirements in reqs are all successfully satisfied; thus, one side-effect of the call is that the negotiator's owner holds exclusive access to all of the resources in s.getResources(), for all s in reqs, when the call returns.
- Negative: The negotiator was not able to satisfy all the requirements in reqs within tout milliseconds; in this case, the call has no effect on the holds set of the negotiator's owner.

A negative response is only possible when tout is not 0. If tout is 0, the call must either return a positive response or not return.

An implementation of trySatisfy(...) must infer, from the requirements it is given and the history kept by the negotiator, an order in which to acquire resources and evaluate conditions without introducing data races. It should also employ a heuristics for avoiding deadlock and starvation and for being fair.

contract(): The closing operation of a prior trySatisfy(...) operation. A negotiator must keep track of the effects of all trySatisfy(...) operations that it has performed and for which it has not yet performed a contract() operation. For brevity, we refer to such trySatisfy(...) operations as being open. When a thread invokes a contract() operation on its negotiator, the negotiator must match the contract() operation to the most

recent trySatisfy(...) operation that it performed and that is still open. The contract() operation must restore the thread's holds set to be the same as it was when the thread invoked this trySatisfy(...) operation. Upon performing the contract() operation this trySatisfy(...) operation is no longer open.

notify(sentry): An instance method for notifying a negotiator that invoking the evaluate-Condition() method of sentry may now return true. The parameter is a special type of sync entry, called a ConditionalSyncSentry, which is used for encoding sync conditions (see Sec. 5.3). The implementation of this method should evaluate sentry.evaluate-Condition() and:

- If sentry.evaluateCondition() returns false or if there is no blocked thread whose negotiator is trying to satisfy sentry, do nothing.
- Otherwise, awaken a blocked thread whose negotiator is trying to satisfy sentry. If there are multiple blocked threads that could be awakened, the negotiator should try to be fair in choosing one.

5.2.3 Reference implementation

We created a reference implementation of the synchronization service library based on the above specifications. Briefly, our library implements the resource numbering synchronization protocol using Java utility locks.[5] To illustrate how we implemented the Negotiator API, we now describe our implementation in detail.

Manage negotiators using thread local storage: Modern operating systems and compilers can statically allocate to a thread a private memory storage (in addition to the thread's stack). Because this storage is not shared and is accessible only by that thread, this

[5] See package java.util.concurrency.locks.

```
1  private static ThreadLocal<SyncNegotiator> negotiator =
     new ThreadLocal<SyncNegotiator>() {
3      protected SyncNegotiator initialValue() {
         return new ReferenceSyncNegotiator();
5      }
     };
7
   static public SyncNegotiator getNegotiator() {
9    return negotiator.get();
   }
```

Figure 5.5: Manage negotiators using thread local storage

storage is called thread local storage (TLS). Therefore, a thread's negotiator object is most suitable to be stored in the TLS. In Java, objects to be stored in the TLS are created using generic class ThreadLocal. Figure 5.5 illustrates how we use this generic class to manage negotiators. Specifically, lines 2–6 create an instance of an anonymous class that inherits from class ThreadLocal¡SyncNegotiator¿.[6] This instance is referenced by a static field negotiator (line 1), which will be used to create thread local negotiators by calling negotiator.get() (line 9). Here, the get() method is supplied by the base class ThreadLocal. The first time a thread calls this method, the method allocates a new negotiator in that thread's TLS and returns that negotiator. Subsequently, the method retrieves the negotiator from the TLS and returns the retrieved negotiator. To allocate a new negotiator, the get() method calls initialValue(). In our implementation, we override this method (lines 3–5) in the anonymous class to return an instance of ReferenceSyncNegotiator—our reference implementation of SyncNegotiator.

Achieve timed locking using Java utility locks: When the tout parameter is greater than 0, our trySatisfy(...) method must measure the time taken for acquiring locks or for

[6]Using an anonymous class is not required for using Java TLS. Here, we use an anonymous class because we need only one instance of that class.

waiting for sync sentries to become true. If the accumulated time exceeds tout, trySatisfy(...) must return immediately. To implement such time measurement, we need to use basic timed locking operations. For example, within a given amount of time, we need to be able to return from an operation that attempts to acquire a lock, regardless whether the lock is successfully acquired or not. Because Java native synchronization primitives (i.e., synchronized statements) do not support timed operations, we instead use Java utility locks in the concurrency package.[7] Specifically, we create an instance of ReentrantLock as the lock for protecting each shared object and call tryLock(tout) on this instance when attempting to acquire the lock. This method returns true when the lock is successfully acquired within tout milliseconds or false otherwise. Additionally, because this lock is explicitly created, we must maintain a mapping from a shared object to the lock that protects this object. As a result, we create a ResourceManager in our synchronization service library to implement this mapping.

Order shared objects using identity hash: Our implementation of the trySatisfy(...) method adopts a common synchronization protocol, called resource numbering, in order to acquire multiple objects without incurring deadlocks. Briefly, this protocol requires that each shared object be assigned a unique number and that all threads agree on a total order of these numbers in acquiring objects. For example, if a thread needs to acquire o_0 and o_1 and if the numbers assigned to these objects are 10 and 6, respectively, then this thread must acquire o_1 (number 6) before acquiring o_0 (number 10), given that the total order that threads have agreed on is the increasing order of these numbers. Because the resource numbering protocol eliminates wait-for cycles, it can effectively avoid deadlocks. However, this protocol is not straightforward to implement. One key challenge is to find a schema of assigning unique numbers to the objects. In C++, the unique number to assign to an object is typically computed from the object's memory

[7] java.util.concurrency.locks

address. In Java, unfortunately, there is no way to obtain an object's address. In fact, even if there were a way to obtain an object's address, it would not be safe to use the address because Java objects typically move around in the JVM as the garbage collector reclaims unused spaces in compacting the memory layout. As a result, an object's address may change; therefore, its assigned number will change, too. In our implementation, we use an object's identity hash directly as the unique number for that object. Intuitively, an identity hash is a sequence number (SN) that is stored in an object's header. An object's SN basically records the number of objects that are created prior to that object—e.g., the first created object has SN 0; the second created object has SN 1, etc. Therefore, this number is unique to each object and will not change as the object moves in the JVM. In Java, the identity hash for an object, *obj*, can be obtained by calling the system function System.identityHashCode(obj). Our implementation also creates some utility functions for comparing and sorting objects based on their identity hashes.

Organize negotiation states using stacks: As mentioned earlier, trySatisfy(...) and contract() are twin methods and must be used pairwise. The negotiator must keep track of what objects are acquired by trySatisfy(...) so that it knows what to release in contract(). To this end, our implementation maintains a stack. Briefly, for each call to trySatisfy(...), the negotiator pushes the set of newly acquired objects (or the empty set, if nothing needs to be acquired) during this call on the top of this stack. For each call to contract(), the negotiator pops off the top set from this stack and releases all the objects in this set. In generating the derived classes (see Section 5.3), our SAP makes sure that these two methods are always called pairwise. Thus, there should be equal number of calls to each method, and the stack should always be empty when a thread terminates.

5.3 Design of our SAP

When the Java compiler encounters a sync-class annotation, it calls a synchronization annotation processor (SAP) to generate a set of derived classes; this set includes a sync controller class and several supporting classes. The synchronization-enhanced program obtained by augmenting an application program with the derived classes generated from the application program's sync classes automatically synchronizes threads as they invoke methods on shared objects.

As explained in Section 5.1, when a thread invokes a method on a controller:

1. If the method's timeout value is greater than zero and the controller does not succeed in enabling the call in time, the controller should return control to the caller without modifying the thread's holds set or executing the method's body.

2. Otherwise, the controller should enable the call; execute the method body; notify other controllers, if any, waiting on a sync condition that might become true as a result of executing the method body; and contract the thread's holds set.

As described in Section 5.2, the controller delegates responsibility to enable the call to a synchronization service, which maintains a negotiator for each thread. The Negotiator API defines the interface between a controller and the negotiator owned by a thread that invokes a method on the controller. The controller uses operations supplied by this negotiator to enable the call and to clean up afterward.

This section describes our reference implementation of a SAP. We describe the derived classes and how they are generated (Sec. 5.3.1), some optimizations that our SAP can perform (Sec. 5.3.2), and the errors in an application program that our SAP will flag (Sec. 5.3.3).

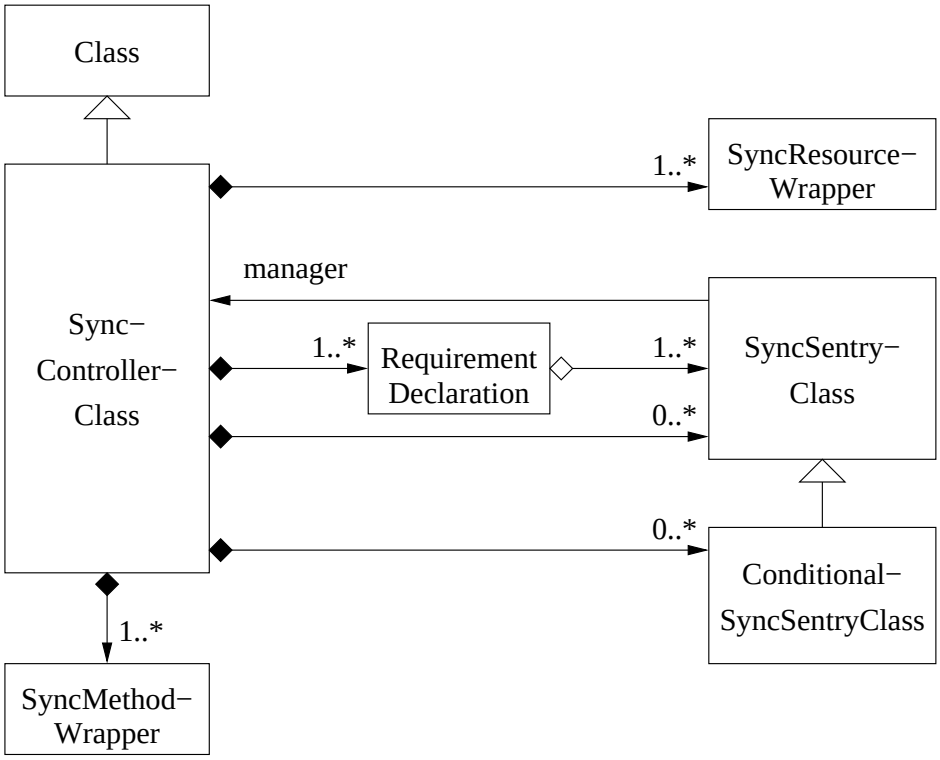


Figure 5.6: Meta Model for the Derived Classes

5.3.1 Generating the derived classes

The meta model in Figure 5.6 depicts the structure of the key derived classes and shows how they fit into the input meta model. Given a program, P, conforming to the input meta model of Figure 5.2, we generate from it a set of classes, P^G, conforming to the meta model of Figure 5.6. Because each class in P^G is derived from one or more synchronization annotations in P, we call these classes the derived classes. The classes in P^G assume a fixed set of supporting classes, P^S. The union of these three sets of classes constitutes a synchronization-enhanced program that implements the functional logic expressed in P while preventing data races and avoiding deadlock. Together, the classes in P^G and P^S encapsulate the synchronization logic.

For simplicity in describing how we generate P^G from P, we assume that the monitor

and optimize values of all sync classes in P are false. The first of these assumptions does not cause any loss in generality, as the monitor property simply provides a convenient short hand for a common programming idiom. Thus, if the monitor value of a sync class, C, in P is true, we just rewrite C, changing the monitor value and strengthening the annotations to indicate that all method bodies conflict with one another. Section 5.3.2 explains how the second of these assumptions affects the generation process in discussing the optimizations that our SAP performs when optimize is true.

We elaborate the meta model for P^G in the sequel, while explaining how P^G is generated. To help organize the remainder of this section, we break it into three divisions: The first division provides an overview of the classes in P^G and their relationship to the constructs in P. The second explains how our SAP generates a sync-controller class. The third indicates how it generates the remaining derived classes.

Overview of the Derived Classes

The dashed arrow in Figure 5.7 depicts the Is-Generated-From relation, showing the types of the declarations in an annotated program, P, that the SAP uses in generating different types of derived classes in the generated program, P^G. Briefly, the SAP generates three types of derived classes from P:

- A unique sync-controller class from a sync class and the sync resources, sync conditions, and sync methods it contains. The sync-controller class encapsulates the code that synchronizes threads that invoke methods on the same sync controller.

- A unique sync-sentry class from a sync method. A sync sentry represents a requirement that a controller must satisfy before executing the body of a sync method or a sync condition.

- A unique conditional sync-sentry class from a sync condition. A special kind of sync

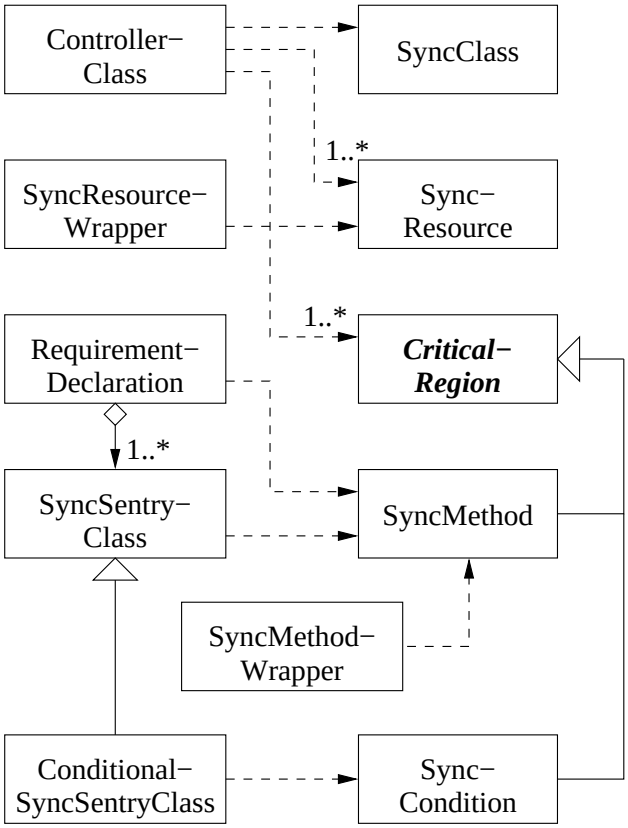


Figure 5.7: The Is-Generated-From Relation

sentry, a conditional sync sentry represents a requirement that a controller must satisfy before executing the body of a special kind of sync method—namely, a sync method with one or more guard conditions (i.e., a sync method with a non-empty guard_conditions value).

We use the remaining meta types in Figure 5.6 for denoting the types of fields and methods that a sync-controller class declares, as described in the sequel.

Generating the Sync-Controller Class

The sync-controller class that our SAP generates from a sync class, C, has the form:

```
final public class SyncClassControllerName extends SyncClassName {
  // Constructor
  // Sync resource wrappers
  // Conditional sync sentries
  // Sync sentries
  // Requirements
  // Sync method wrappers
}
```

where SyncClassName denotes C's name, SyncClassControllerName denotes the name designated by C's controller property, and the comments stand for the different kinds of declarations described in the sequel. For example, the sync-controller class generated from the BallRoom class (Fig. 5.3) has the form shown in Figures 5.8 and 5.9.

The comments in this code skeleton stand for declarations of the following:

Constructor: A public constructor, which simply delegates to C's constructor. Line 2 of Figure 5.8 illustrates this construction.

Sync resource wrappers: A sync-resource wrapper, for each sync resource, R, in C. The generated declaration depends on whether R is mutable (i.e., the mutable property is true) or immutable (i.e., the mutable property is false). If R is mutable, the generated declaration has the form:

```
final public Object GeneratedFieldName
  = new SyncResourceWrapper(SyncResourceFieldName);
```
(5.1)

where SyncResourceFieldName is the field name declared in R and GeneratedField-

```
final public class BallRoomSync extends BallRoom {
  public BallRoomSync() { super(); }

  final public Object boysResource
    = new SyncResourceWrapper(boys);
  final public Object girlsResource
    = new SyncResourceWrapper(girls);

  final public ConditionalSyncSentry
     hasBoysConditionalSyncSentry
    = new HasBoysConditionalSyncSentry(this);
  final public ConditionalSyncSentry
     hasGirlsConditionalSyncSentry
    = new HasGirlsConditionalSyncSentry(this);

  final public SyncSenty addBoySyncSentry
    = new AddBoySyncSentry(this);
  final public SyncSentry addGirlSyncSentry
    = new AddGirlSyncSentry(this);
  final public SyncSentry pairSyncSentry
    = new PairSyncSentry(this);

  final public SyncSentry addBoyRequirement[]
    = {addBoySyncSentry};
  final public SyncSentry addGirlRequirement[]
    = {addGirlSyncSentry};
  final public SyncSentry pairRequirement[]
    =  {pairSyncSentry,
        hasBoysConditionalSyncSentry,
        hasGirlsConditionalSyncSentry};

  public void addBoy(Person boy) { ... }   // See next
     figure

  ...
}
```

Figure 5.8: An Example Synchronization Controller Class

```
1  final public class BallRoomSync extends BallRoom {

3    ...  // See previous figure

5    final public void addBoy(Person boy) {
       SyncNegotiator neg = SyncNegotiator.getNegotiator();
7      try {
         try {
9          neg.trySatisfy(addBoyRequirement, 0);
           super.addBoy(boy);
11       }
         finally {
13         neg.contract();
         }
15       neg.notify(hasBoysConditionalSyncSentry);
       }
17     catch (Exception e) {
         e.printStackTrace();
19     }
     }
21   ...
}
```

Figure 5.9: An Example Sync Method Wrapper

Name is a new field name. If R is immutable, the generated declaration has the form:

$$\begin{aligned} &\text{final public Object GeneratedFieldName} \\ &\quad = \text{new SyncResourceWrapper();} \end{aligned} \tag{5.2}$$

where GeneratedFieldName is a new field name. By convention, assuming no name conflicts, we generate the new field name by appending the string "Resource" to the end of the former field name.[8] Lines 4–7 of Figure 5.8 show the declarations for the sync-resource wrappers generated from the declarations for boys and girls in the

[8]While any new name will suffice, following conventions in naming makes the generated code much easier to read.

BallRoom class (Fig. 5.3).

Class SyncResourceWrapper is defined in P^S. It has two constructors.

1. SyncResourceWrapper(obj) creates a delegate object for obj. This delegate object overrides its hashCode() method by returning the identity hash for obj. We call obj as the delegate object's base object.
2. SyncResourceWrapper() creates a delegate object for a new base object. It is equivalent to SyncResourceWrapper(new Object()).

As mentioned in Section 5.2.3, our synchronization service library maintains a hash map (i.e., an instance of HashMap) to associate a sync resource with a lock. Because a mutable object cannot be used as a key of a hash map[9], our SAP creates a delegate object for each mutable sync resource using constructor (1). Subsequently, our synchronization service library uses this delegate object as the key when querying the lock associated with a sync resource. Constructor (2) is used to create a delegate object for an immutable sync resource. Due to the effect of Java autoboxing, an immutable sync resource changes its value by referencing a different immutable object. Because these objects have different identity hash values, the hashCode() method of the delegate object created for an immutable sync resource should not depend on what the immutable sync resource references. Therefore, the delegate object that our SAP creates for an immutable sync resource uses a new base object, whose identity hash remains constant as the immutable sync resource changes the object it references.

However, if an immutable sync resource is also multi-referenced (i.e., the multiref property is true), using a new base object can greatly complicate code generation. On the one hand, being multi-referenced means that there exist multiple sync resources

[9]According to Java HashMap's implementation, an object can be used as a key only if the value returned by the object's hashCode() method does not depend on the object's content.

referencing the same object *obj*. Therefore, the delegates for these sync resources must use the same base object (otherwise, *obj* will be associated with different locks). On the other hand, being immutable means that the base object used by the delegates must be newly created, rather than *obj* itself. Therefore, our SAP must create this base object first and then use this base object to create the delegates, possibly using constructor (1). Because these delegates may belong to different sync classes, passing this base object across different sync classes may require changes to the signatures of some existing methods. For purposes of expediency, therefore, we do not permit immutable and multi-referenced sync resources in the current version of our SAP; in other words, we do not permit the mutable property to be false and the multiref property to be true at the same time.

Conditional sync sentries: A conditional sync sentry, for each sync condition, S, in C. Intuitively, this field gives the controller a reference to a conditional sync sentry representing a requirement that must be satisfied before executing the method body of any method for which S is a guard condition. The generated declaration has the form:

```
final public ConditionalSyncSentry GeneratedFieldName
  = new GeneratedClassName(this);
```
(5.3)

where GeneratedFieldName and GeneratedClassName are both names generated by our SAP. By convention, assuming no name conflicts, we generate the former name by appending the string "ConditionalSyncSentry" to S's name and the latter name by converting the first letter of the former name to upper case. The latter name is then also used as the class name in the conditional sync-sentry class that is generated from S (described later in this subsection). ConditionalSyncSentry is the base class, defined in P^S, for all other conditional sync-sentry classes. Lines 9–12 of Figure 5.8

show the declarations for the conditional sync sentries generated from the declarations for hasBoys and hasGirls in the BallRoom class (Fig. 5.3).

Sync sentries: A sync sentry, for each sync method, M, in C. Intuitively, this field gives the controller a reference to a sync sentry representing a requirement that a negotiator must satisfy to enable execution of the body of M—namely, the requirement that the negotiator's owner needs exclusion on the resources indicated in M's resources property. The generated declaration has the form:

$$\begin{aligned} &\text{final public SyncSentry GeneratedFieldName} \\ &\quad = \text{new GeneratedClassName(this);} \end{aligned} \tag{5.4}$$

where GeneratedFieldName and GeneratedClassName are both names generated by our SAP. By convention, assuming no name conflicts, we generate the former name by appending the string "SyncSentry" to M's name and the latter name by converting the first letter of the former name to upper case. The latter name is then also used as the class name in the sync-sentry class that is generated from M (described later in this subsection). SyncSentry is the class, defined in P^S, for all other non-conditional sync-sentry classes. Lines 14–19 of Figure 5.8 show the declarations for the sync sentries generated from the declarations for addBoy, addGirl, and pair in the BallRoom class (Fig. 5.3).

Requirements: An array of sync sentries (including conditional sync sentries), for each sync method, M, in C. Intuitively, this field gives the controller a reference to the full list of requirements that a negotiator must satisfy in order to enable a call to M. The

generated declaration has the form:

$$\begin{aligned} &\text{final public SyncSentry GeneratedFieldName[]} \\ &\quad = \{\ \text{SelfSyncSentry [, GuardConditionalSyncSentries] }\}; \end{aligned} \tag{5.5}$$

where GeneratedFieldName is a name generated by our SAP, SelfSyncSentry is the field name of M's sync sentry generated in (5.4), and GuardConditionalSyncSentries is a list of comma-separated field names generated using (5.3) from M's guard conditions. By convention, assuming no name conflicts, we generate GeneratedFieldName by appending the string "Resources" to M's name. Lines 21–28 of Figure 5.8 show the declarations for the requirements generated from the declarations for addBoy, addGirl, and pair in the BallRoom class (Fig. 5.3).

Sync method wrappers: A wrapper method for each sync method, M, in C. Briefly, the wrapper obtains the negotiator, n, that belongs to the calling thread and invokes n.trySatisfy(...), passing it the requirements generated for M (described previously). If the negotiator is successful, the wrapper delegates to C's version of M (executing M's body), and then invokes n.contract() to contract the calling thread's holds set and n.notify(...) with references to the conditional sync sentries representing the requirements that might be impacted; otherwise, it just invokes n.contract() and returns.

In more detail, the generated declaration has the form:

```
final public ReturnType MethodName (FormalParametersDecl) {
  SyncNegotiator neg = SyncNegotiator.getNegotiator();
  AdditionalVarDecl
  try {
    try {
      NegotiateStatement
      SuperSendStatement
    }
    finally {
      neg.contract();
    }
    NotifyList
  }
  catch (Exception e) {
    e.printStackTrace();
  }
  ReturnStatement
}
```

where ReturnType, MethodName, and FormalParametersDecl are M's return type, M's method name, and M's formal parameters declaration, respectively. NotifyList has the form:

```
[ neg.notify(ImpactConditionalSyncSentry); ]*
```

where ImpactConditionalSyncSentry is the field name generated in Figure 5.3 from the declaration of one of M's impact conditions, if any. The forms of AdditionalVarDecl, SuperSendStatement, and ReturnStatement depend on whether M's return type is

void or not. If it is void, then AdditionalVarDecl and ReturnStatement are empty; SuperSendStatement has the form:

super.MethodName(ParametersNames);

where ParametersNames is a comma-separated list of M's formal parameter names. Otherwise, if M's return type is not void, then AdditionalVarDecl has the form:

ReturnType ret;

SuperSendStatement has the form:

ret = super.MethodName(ParametersNames);

And ReturnStatement has the form:

return ret;

Going back to the declaration of sync method wrappers, the form of NegotiationStatement depends on whether the timeout property of M's annotation is greater than 0 or not. If timeout is 0 or is not given (default is 0), then NegotiationStatement has the form:

neg.trySatisfy(SelfSyncSentry, 0);

where SelfSyncSentry is the field name generated in Figure 5.3 from M. Otherwise, if timeout is greater than 0, then NegotiationStatement has the form:

```
SyncNegotiatorResponse resp =
    neg.trySatisfy(SelfSyncSentry, TimeoutValue);
if (resp.isNegative()) ReturnStatement
```

where TimeoutValue is the value of the timeout property. Figure 5.9 shows the declaration for the sync method wrapper generated from the declaration for addBoy in the BallRoom class (Fig. 5.3).

Generating the Sync-Sentry Classes

It remains to describe how the sync-sentry classes are generated from the declarations of sync conditions and sync methods. Recall from Section 5.2 that, when consulting a negotiator, a sync controller passes the negotiator an array of sync sentries and a timeout value. Each sync sentry in the array represents a requirement that the negotiator must try to satisfy, where the requirement expressed by a sync sentry, s, is that the negotiator's owner must have exclusive access to all resources returned by s.getResources() and s.evaluateCondition() must return true. To implement these methods, a sync sentry has a field named manager (Fig. 5.6) that references the sync controller that aggregates it. This field allows the sync sentry to construct the handles (i.e., references to sync-resource wrappers) that getResources() returns and to evaluate a sync condition.

The class generated by our SAP from a sync method, M, has the form:

```
final public class GeneratedClassName extends SyncSentry {
  private SyncControllerName manager = null;
  public GeneratedClassName (SyncControllerName manager) {
    this.manager = manager;
  }
  @Override
  public Object[] getResources() {
    Object handles[] = { Handles };
    return handles;
  }
}
```
(5.6)

where GeneratedClassName is the class name generated from M by (5.4); SyncControllerName is the name specified by the controller property of the sync class that contains M; and Handles is a comma-separated list of expressions corresponding to the sync resources named in M's resources property; each of these expressions has the form

manager.GeneratedFieldName

where GeneratedFieldName is the name generated by (5.1) from one of the aforementioned sync resources. The base class SyncSentry, defined in P^S, defines evaluateCondition() to always return true. Because the sync-sentry class generated by (5.6) encodes the requirement embodied by M's resources property, not by one of M's guard conditions, the sync-sentry class just inherits this definition of evaluateCondition().

For example, Figure 5.10 shows the sync-sentry class that our SAP generates from the declaration of pair() in the BallRoom class (Fig. 5.3). Recall that when a BallRoomSync

```
final public class PairSyncSentry extends SyncSentry {
  private BallRoomSync manager = null;

  public PairSyncSentry(BallRoomSync manager) {
    this.manager = manager;
  }

  @Override
  public Object[] getResources() {
    Object handles[]
      = {manager.girlsResource, manager.boysResource};
    return handles;
  }
}
```

Figure 5.10: An Example Sync-Sentry Class

controller creates an instance of PairSyncSentry (Fig. 5.8, lines 18–19), it passes the constructor a self reference. Thus, the constructor initializes manager (Fig. 5.10, lines 4–6) with a reference to the right controller. Additionally, getResources() returns references to the sync-resource wrappers generated from the declarations of boys and girls, the sync-resources designated by pair()'s resources property (lines 9–12).

The conditional sync-sentry class generated from the declaration of a sync condition, S, is similar, the main difference being that it redefines evaluateCondition() to delegate to S.

It has the form:

```
final public class GeneratedClassName extends ConditionalSyncSentry {
  private SyncControllerName manager = null;
  public GeneratedClassName (SyncControllerName manager) {
    this.manager = manager;
  }
  @Override
  public Object[] getResources() {
    Object handles[] = { Handles };
    return handles;
  }
  @Override
  public boolean evaluateGuard() {
    return manager.SyncConditionName();
  } }
```
(5.7)

where GeneratedClassName is the class name generated from S by (5.4); SyncControllerName is the name specified by the controller property of the sync class that contains S; and Handles is a comma-separated list of expressions corresponding to the sync resources named in S's resources property; each of these expressions is as described previously for a sync-sentry class.[10] Figure 5.11 illustrates this construction. It is generated from the definition of hasBoys() in the BallRoom class (Fig. 5.3).

[10]We will consider a sync sentry's arguments in the next version.

```
final public class HasBoysConditionalSyncSentry
    extends ConditionalSyncSentry {
  private BallRoomSync manager = null;

  public HasBoysConditionalSyncSentry(BallRoomSync manager)
     {
    this.manager = manager;
  }

  @Override
  public boolean evaluateGuard() {
    return manager.hasBoys();
  }

  @Override
  public Object[] getResources() {
    Object handles[] = {manager.boysSyncResource};
    return handles;
  }
}
```

Figure 5.11: An Example ConditionalSyncSentry Class

5.3.2 Optimizations

We say two methods conflict with each other, if they need the same sync resource. These methods are thus called conflicting methods. The goal of thread synchronization is to prevent conflicting methods from executing concurrently on the same object. Therefore, any optimization must preserve this goal. Our SAP may perform three optimizations in generating the getResources() method of a sync sentry class. All the optimizations aim to reduce the number of objects that getResources() returns while still guaranteeing that if a sync controller receives calls on conflicting methods, it serializes the method executions. We now describe these optimizations in the order that our SAP performs them.

#1: Ignore a sync resource needed by a sync method, if the sync resource is also needed by

Table 5.3: Sync resource needs of a hypothetical sync class

Method Name	Required Sync Resources
m_0	r_0, r_1
m_1	r_3, r_4
m_2	r_3, r_4, r_5
m_3	r_1
m_4	r_2

the sync method's guards. As described earlier, when a thread executes a sync method, the thread's negotiator guarantees that the sync method has exclusive access to all the objects designated by the following sync resources.

1. Those specified in the resources property of the sync method's annotation.

2. Those specified in the resources property of a sync condition's annotation, if the sync condition is one of the sync method's guards.

Therefore, if a sync resource is included in both cases above, then we can safely remove that sync resource from case (1) without changing the set of objects that a negotiator will acquire.

#2: Remove sync resources that appear only once. If a sync resource, R, is needed by only one method, M, then R can be the cause of a conflict between two executions of methods on the same object, only if the methods are both M. We call such a sync resource a lonely sync resource. The significance of a lonely sync resource is that it can be removed from the set of resources a negotiator needs to acquire before executing M, provided that M needs at least one other resource. However, because a method may conflict with itself—for example, when multiple threads concurrently invoke the same method—a lonely sync resource should not be removed, if it is the only sync resource that the method needs. For example, assume Table 5.3 lists the sync resources needed by each of the sync methods and sync conditions of some sync class. Here, r_0, r_5, and r_2 are lonely sync resources; however, only r_0 and r_5 can

Table 5.4: Sync resource needs of a conceptual sync class after optimization #2

Method Name	Required Sync Resources
m_0	r_1
m_1	r_3, r_4
m_2	r_3, r_4
m_3	r_1
m_4	r_2

Table 5.5: Sync resource needs of a hypothetical sync class after optimization #3

Method Name	Required Sync Resources
m_0	r_1
m_1	r_3
m_2	r_3
m_3	r_1
m_4	r_2

be safely removed from the result that getResources() returns; r_2 should not be removed, because it is the only sync resource needed by m_4. Table 5.4 illustrates the sync-resource needs after our SAP performs this optimization.

#3: Use one sync resource from each equivalent set of sync resources. We say that two sync resources are synchronization-equivalent, if every sync method/condition either needs both of the resources or does not need either of them. The significance of this equivalence relation is that it suffices to use only one sync resource as a representative of an entire synchronization-equivalence class in order to synchronize conflicting methods. For example, Table 5.4 indicates that r_3 and r_4 are equivalent, because they are both needed by m_1 and m_2 and because none of them is needed by other methods. Therefore, we can keep only one sync resource from this set without failing to synchronize the conflicting methods m_1 and m_2. Suppose we choose to keep r_3 and remove r_4, then Table 5.5 illustrates the sync resource needs after our SAP performs this optimization.

If the optimized property of @SyncClass is given value true, our SAP performs the above

Table 5.6: Some usage constraints checked by our SAP

Annotation	Required Modifier	Forbidden Modifier	Return Type
@SyncClass	abstract	final	N.A.
@SyncResource	none	private, public*	N.A.
@SyncCondition	public	abstract, synchronized*	boolean
@SyncMethod	none	abstract, final, private, synchronized*	any

optimizations before generating the getResources() methods.

5.3.3 Error processing

To help a programmer write consistent and meaningful synchronization annotations, our SAP checks potential errors before generating the derived classes. An error can be serious, causing the SAP to abort, or mild producing only a warning. We now list the errors that our SAP checks.

Serious errors: A serious error indicates that the generated code may not compilable. Because such an error will result in unusable code, we abort the code generation process as soon as a serious error surfaces. Table 5.6 gives a summary of some usage constraints that our SAP checks. A violation of any of the constraints (that are not marked by a "*") is considered a serious error. For example, if a programmer uses @SyncClass to annotate a final class, then the derived class our SAP generates from it will not compile, because, in Java, a final class cannot be extended. In addition to checking these constraints, our SAP also checks the following serious errors.

- A sync class does not contain a factory method.
- The mutable property of a sync resource is false and the multiref property of that sync resource is true (for reasons explained in Section 5.3.1).

- A sync condition throws exceptions, because throwing exceptions may make side effects.

Mild errors: A mild error indicates that the compiled byte code may have flaws or may not be safe to execute. Upon detecting a mild error, we simply print out a warning message, rather than aborting the code generation process. The three constraints that are marked by "*" in Table 5.6 will lead to mild errors. For example, although declaring a public sync resource will not prevent the derived classes from being compiled, being public may grant the clients of the sync class permission to directly modify that sync resource, which may not be safe. In addition to these mild errors, our SAP also checks for the following mild errors.

- A sync resource that is not contained in the resources property of any sync method or any sync condition.
- A sync condition that is not contained in the guard_conditions property of any sync method.
- A sync condition that is not contained in the impact_conditions property of any sync method.
- A sync resource that belongs to a monitor sync class (i.e., the monitor property is true). We call such a sync condition an orphan condition, because it may never become true. Thus, the sync method that relies on this condition may never be able to execute.

5.4 Survey of classical concurrency problems

We did a survey of using our generative approach in order to investigate the following questions:

- How will use of synchronization contracts help with the synchronization problem raised in Chapter 2?
- Are synchronization annotations sufficiently expressive to describe typical synchronization scenarios?

To gain insight into the first question, we revisit the problem of extending the ballroom example and compare a solution using synchronization contracts with that presented in Chapter 2. The second question requires us to apply our generative approach on a broader range of synchronization scenarios. This section describes the survey we performed in approaching these questions.

5.4.1 Handling the extension problem

In Chapter 2, we pointed out four deadlock cases that can arise when we extend the BallRoom class with a waiting room. Specifically, the first two deadlock cases come into being, because the mixing of the synchronization code with the functional code imposes additional constraints on how the new synchronization code can be added to the existing code. For example, because a programmer must make sure that the waiting room has vacancies before adding a boy to the waiting room, the programmer must first acquire the lock that protects the waiting room before acquiring the lock that protects the boys queue. The last two deadlock case come into being, because of a flaw in the synchronization logic—so long as the waiting room allows the room to fill up with all boys or all girls, the deadlock will occur. We now describe how use of synchronization contracts can help eliminate these deadlock cases.

Using synchronization contracts avoids the first two deadlock cases, simply because the synchronization logic is separated from the functional logic. Such separation eliminates the constraints of implementing the synchronization logic. Because our synchronization annotations do not require any specific order in acquiring sync resources, the synchronization

```
1  @SyncClass(controller = "BallRoomSync", monitor = true)
   abstract public class BallRoom {
3    private Queue<Person> boys = new LinkedList<Person>();
     private Queue<Person> girls = new LinkedList<Person>();
5    private final int SIZE = 10;
     private int numChildrenWaiting = 0;
7
     static public BallRoom instance() { return new
        BallRoomSync(); }
9
     @SyncMethod(guard_conditions = {"hasWaitingRoomSpace"},
11                 impact_conditions = {"hasBoys"})
     public void addBoy(Person boy) {
13     boys.add(boy);
       ++numChildrenWaiting;
15   }

17   @SyncMethod(...)
     public void addGirl(Person girl) { ... }
19
     @SyncMethod(guard_conditions = {"hasBoys", "hasGirls"},
21                 impact_conditions={"hasWaitingRoomSpace"})
     public void pair() {
23     boys.remove();
       girls.remove();
25     numChildrenWaiting -= 2;
     }
27
     @SyncCondition
29   public boolean hasBoys() { return boys.size() > 0; }

31   @SyncCondition
     public boolean hasGirls() { return girls.size() > 0; }
33
     @SyncCondition
35   public boolean hasWaitingRoomSpace() {
       numChildrenWaiting < SIZE;
37   }
   }
```

Figure 5.12: Implementation of BallRoom extended with a waiting room using synchronization annotations (fixed the first two deadlock cases)

service that a programmer chooses to use is free to adopt any order at runtime to avoid deadlocks, be fair, and be efficient. For instance, Figure 5.12 illustrates how we extend the annotations in the BallRoom class (Figure 5.3) to address the additional synchronization needs required by the waiting room extension. Because methods addBoy(...), addGirl(...), and pair() all modify field numChildrenWaiting, we create a monitor controller by setting the monitor property of the @SyncClass annotation to true (line 1). As a result, we no longer need to annotate the fields as sync resources nor need to assign values to the resources property of the @SyncMethod and the @SyncCondition annotations. We introduce a new sync condition hasWaitingRoomSpace() (lines 34–37) and refer to this sync condition as the guard for addBoy(...) (line 10) and addGirl(...) (line 17). Because pair() decrements numChildrenWaiting, this sync condition is also an impact condition of pair() (line 21). That is all what we need to do in extending the annotations to support the waiting room extension. Compared with Java synchronization code (Figure 2.4), using synchronization annotations makes the extension much easier and less prone to errors. Both the synchronization annotations and the functional code are more readable than the mixing of the synchronization code with the functional code.

Fixing the other two deadlock cases requires a redesign of the synchronization logic. Synchronization annotations can provide analytical insight into the logic flaws in the synchronization logic. An example is a (directed) synchronization-dependency graph, which reveals the relationships between each pair of a sync resource, a sync method, and a sync condition. Specifically, an edge from a sync resource to a sync method (or sync condition) signifies that the sync resource is contained in the resources property of the sync method's (or sync condition's) annotation; an edge from a sync condition to a sync method signifies that the sync condition is a guard condition of that sync method; an arrow from a sync method to a sync condition signifies that the sync condition is an impact condition of that sync method. Therefore, a dependency graph can be generated solely based on the syn-

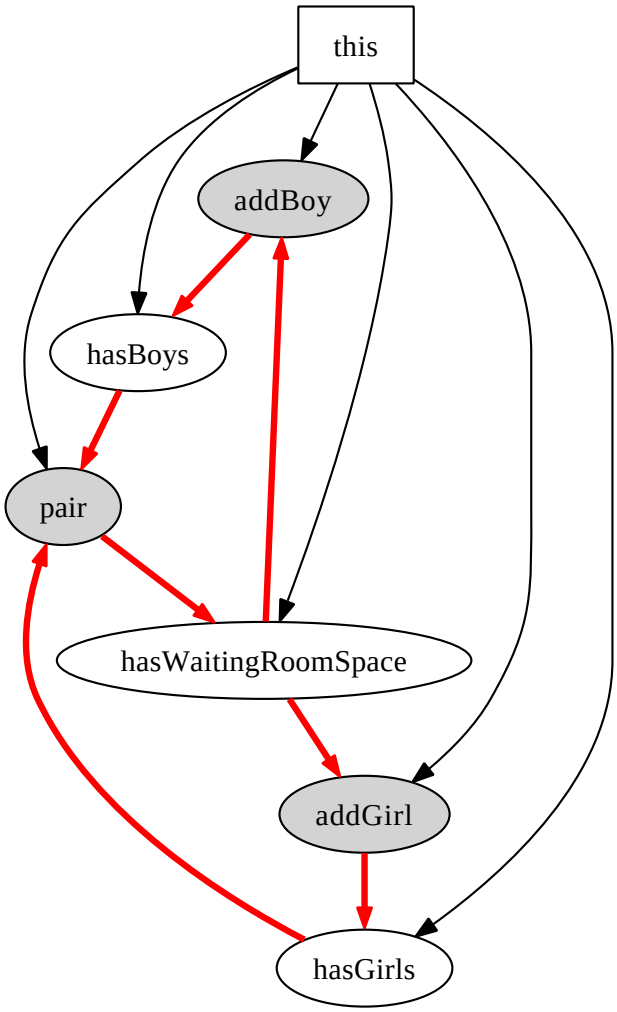


Figure 5.13: Dependency graph of Figure 5.12. For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.

chronization annotations. For example, Figure 5.13 illustrates the dependency graph of the synchronization annotations in Figure 5.12. In this graph, the rectangle represents the current monitor object; shaded ellipses represent sync methods; hollow ellipses represent sync conditions. The two highlighted cycles (red and bold) in this graph indicate the other two deadlock cases. As these two cycles are essentially the same, let's focus on only one of them:

hasGirl → pair → hasWaitingRoomSpace → addGirl → hasGirl

Intuitively, in order to execute a sync method on this cycle (i.e., pair() or addGirl(...)), the

sync condition to the left of the sync method must be true. Because these sync conditions are, in turn, impacted by the sync methods to their left side, deadlock can occur if all the sync methods on this cycle are blocked at the same time. Therefore, in order to block all these sync methods at the same time, all the sync conditions on this cycle must be false at the same time. A false value for hasGirls() signifies that there is no girl, while a false value for hasWaitingRoomSpace() signifies that the waiting room is full. The only situation when the waiting room is full but there is no girl is when the waiting room is full of boys, which is likely to happen. Therefore, when the waiting room is full of boys, a deadlock will occur. Similarly, the other cycle indicates the other deadlock case, when the waiting room is full of girls. As a consequence, in order to avoid such deadlock, the programmer must prevent the number of boys or the number of girls in the waiting room from reaching the maximum capacity of the room. Implementing such logic can be very complex and error-prone using Java synchronization primitives; however, implementing such logic using synchronization annotations is mush easier. Figure 5.14 illustrates one possible implementation. Specifically, we introduce two new sync conditions, willNotBeAllBoys (lines 26–29) and willNotBeAllGirls (lines 31–34). The former condition returns true if the waiting room will not be filled up with all boys after adding an additional boy. This condition is a guard (in addition to hasWaitingRoomSpace) of addBoy(...) (line 8). Likewise, the latter condition returns true if the waiting room will not be filled up with all girls after adding an additional girl and is a guard of addGirl(...) (line 12, elided). Both conditions are impact conditions of pair() (lines 17 and 18). In order to implement these conditions, we need to keep track of the number of waiting boys and the number of waiting girls separately. Therefore, we split numChildrenWaiting into two variables numBoysWaiting (line 4) and numGirlsWaiting (line 5).

```
@SyncClass(controller = "BallRoomSync", monitor = true)
abstract public class BallRoom {
  // Lines 3--5 and 8 of Figure 5.12
  private int numBoysWaiting = 0;
  private int numGirlsWaiting = 0;

  @SyncMethod(guard_conditions = {"hasWaitingRoomSpace",
                                  "willNotBeAllBoys"},
              impact_conditions = {"hasBoys"})
  public void addBoy(Person boy) { ... }

  @SyncMethod(...)
  public void addGirl(Person girl) { ... }

  @SyncMethod(guard_conditions = {"hasBoys", "hasGirls"},
              impact_conditions={"hasWaitingRoomSpace",
                                 "willNotBeAllBoys",
                                 "willNotBeAllGirls"})
  public void pair() { ... }

  @SyncCondition
  public boolean hasWaitingRoomSpace() {
    numBoysWaiting + numGirlsWaiting < SIZE;
  }

  @SyncCondition
  public boolean willNotBeAllBoys() {
    return numBoysWaiting + 1 < SIZE;
  }

  @SyncCondition
  public boolean willNotBeAllGirls() {
    return numGirlsWaiting + 1 < SIZE;
  }

  // Lines 28--32 of Figure 5.12
  ...
}
```

Figure 5.14: Implementation of BallRoom extended with a waiting room using synchronization annotations (fixed all deadlock cases)

5.4.2 Results of implementing 24 classical concurrency problems

The other question we wanted to investigate through our survey is whether it is possible to use synchronization annotations to describe typical synchronization scenarios. To find an answer to this question, we need a broader spectrum of examples that represent various concurrency problems. For this reason, we applied our generative approach on 24 classical synchronization problems, 15 of which cover all the synchronization problems in Magee's book Concurrency: State Models & Java Programs (2nd) [51].

Table 5.7 gives a list of the subject problems. The first 15 are from Magee's book. The problems without sync resources are implemented as monitors. The problems without sync conditions use no condition synchronization. The numbers in this table indicate that, although the number of sync methods in these programs range from 1 to 8, the synchronization logic of most programs can be expressed using no more than 2 sync resources and 2 sync conditions. The fair single-lane bridge program, which has 5 sync resources, is one exception. However, our SAP is able to optimize this number to 3. The snapshot player, which has 3 sync resources and 3 sync conditions, is the other exception. Again, our SAP can optimize the number of resources to 2. The source code of these programs and the code that our SAP generates from these programs can be found at our website.[11]

Generally speaking, we were able to come up with elegant implementations for all these problems using synchronization annotations. For the 15 problems in Magee's book, we were able to derive our implementations by reusing most of the given functional code and replacing the given synchronization code with synchronization annotations. Specifically, for each problem in the book, we mapped each Java method (excluding constructors) given in the book to a sync method. If this method uses wait() inside a while loop, we created a sync condition to encapsulate the boolean expression of the loop.[12] We then referred to this

[11] http://www.cse.msu.edu/sens/szumo/madura

[12] but reverse the logic. For example, a boolean expression in a while loop signifies "when to

Table 5.7: Subject problems

		Number of sync		
Problem name	classes	resources	conditions	methods
ornamental garden	1	1	0	2
car park	1	1	2	2
semaphores	1	0	1	2
bounded buffer	1	1	2	3
printer & scanner	1	2	0	2
dining philosophers	1	2	0	1
single-lane bridge	1	2	2	4
single-lane bridge (fair)	1	5	2	6
single-lane bridge (fair + optimized)	1	3	2	6
single-lane bridge (monitor)	1	0	2	6
readers & writers	1	0	2	4
golf club	1	0	1	2
golf club (fair)	1	0	2	4
async port	1	0	0	2
linda tuple space	1	0	1	6
ball room	1	2	2	3
sleeping barber	1	2	2	7
barrier	1	1	1	2
bath room	1	2	2	4
ping pong	1	1	2	2
river	1	2	2	3
nested monitors	2	2	0	4
snapshot player	1	3	3	8
snapshot player (optimized)	1	2	3	8

```
public class Buffer<E> {
  protected Queue<E> buffer = new LinkedList<E>();
  protected int size = 0;

  public Buffer(int size) {
    this.size = size;
  }

  public synchronized void put(E o)
      throws InterruptedException {
    while (buffer.size() == size) wait();
    buffer.add(o);
    notifyAll();
  }

  public synchronized E get()
      throws InterruptedException {
    while (buffer.size() == 0) wait();
    E o = buffer.remove();
    notifyAll();
    return o;
  }
}
```

Figure 5.15: Implementation of a bounded buffer class in Magee's book

sync condition as a guard condition of the sync method. If the Java method uses notify() or notifyAll(), we first found out the sync conditions whose values might become true after executing the sync method; and then listed these sync conditions as the impact conditions of the sync method. We also marked the shared fields as sync resources and referred to them appropriately in the annotations.

Figure 5.15 shows the implementation of a bounded buffer class given in Magee's book. For purpose of illustration, we slightly modified this implementation so that it uses a Java Queue to simulate a buffer, rather than emulating the buffer using a primitive array and sev-

wait", whereas a sync condition signifies "when to proceed".

eral integers. Figure 5.16 shows an implementation that uses synchronization annotations. Briefly, the methods and constructor in Figure 5.15 are directly copied to Figure 5.16, except that the synchronization code is removed and that the synchronization logic is specified in the synchronization annotations. Additionally, the sync methods do not need to throw InterruptedException, because this exception will be automatically handled by the synchronization service. Moreover, two sync conditions (lines 29–37) are introduced to encapsulate the boolean expressions (lines 11 and 18) in the original implementation.

Although mapping sync methods is mechanical, creating sync conditions requires us to first understand the synchronization logic buried in the original implementation. However, it allows us to extract important information about the synchronization logic and to document such information explicitly using synchronization annotations. In fact, software engineering experts have recommended conditionals be encapsulated into methods with meaningful names [53]. Our generative synchronization approach can enforce this coding style to some extent. Additionally, we believe that associating synchronization contracts with methods, rather than code blocks, allows some useful software engineering benefits. First, it can facilitate program comprehension due to the absence of block-level synchronization. Second, it can enforce creation of short and logically coherent methods, because code with different synchronization requirements must belong to different sync methods. Software engineering experts also recommend writing short methods to improve code robustness and maintainability [53, 52, 19]. Our generative synchronization approach is also compatible with this recommended practice.

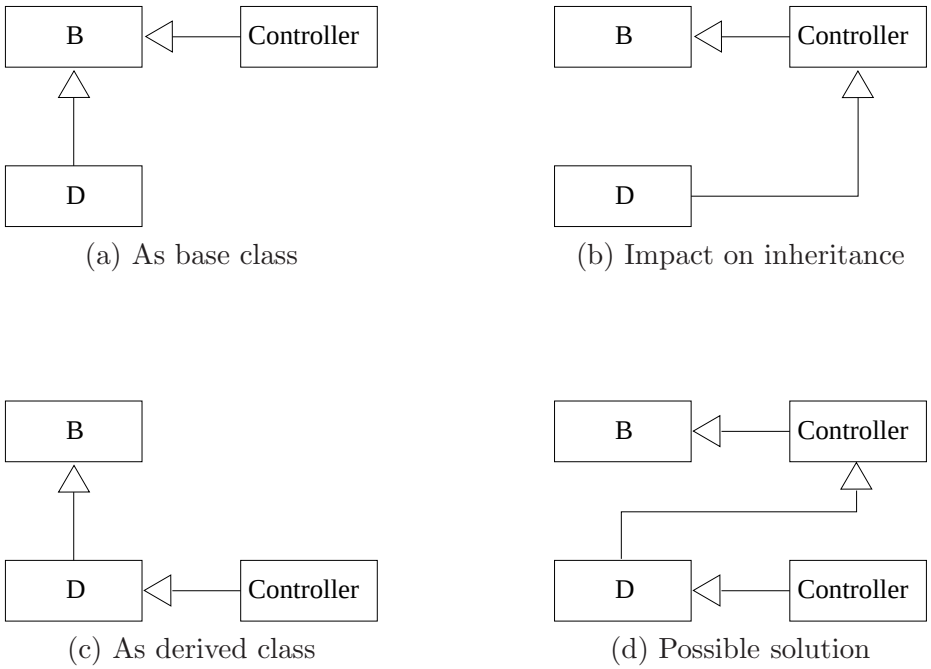
5.5 Known limitations

We have realized some limitations to our generative approach. One limitation is that our approach cannot handle the deadlock cases that may arise when a sync method is called inside

```
1  @SyncClass(controller = "BufferSync", monitor = true)
   abstract public class Buffer<E> {
3    protected Queue<E> buffer = new LinkedList<E>();
     protected int size = 0;
5
     static public Buffer instance(int size) {
7      return new BufferSync(size);
     }
9
     public Buffer(int size) {
11     this.size = size;
     }
13
     @SyncMethod(guard_conditions = {"hasSpace"},
15               impact_conditions = {"hasData"})
     public void put(E o) {
17     buffer.add(o);
     }
19
     @SyncMethod(guard_conditions = {"hasData"},
21               impact_conditions = {"hasSpace"})
     public E get() {
23     return buffer.remove();
     }
25
     @SyncCondition
27   public boolean hasSpace() {
       return buffer.size() < size;
29   }

31   @SyncCondition
     public boolean hasData() {
33     return buffer.size() > 0;
     }
35 }
```

Figure 5.16: Implementation of a bounded buffer class using synchronization annotations



(a) As base class (b) Impact on inheritance

(c) As derived class (d) Possible solution

Figure 5.17: Sync classes in a class hierarchy

of another sync method. Because each of these sync methods may need different sets of sync resources, calling a sync method inside of another sync method will lead to incremental locking. For example, if sync method M_0 needs sync resource r_0, then when a thread calls M_0, the thread will try to acquire r_0. During executing M_0, if the thread calls another sync method M_1, which needs r_1, then the thread will try to acquire r_1 while holding r_0. A deadlock may occur if another thread calls M_0 inside of M_1 or calls another sync method M_2 that will try to acquire r_0 while holding r_1. As mentioned in Section 4.2, such deadlock cases can be avoided by composing M_0's contract with M_1's contract. Therefore, as the thread calls M_0, the thread will acquire r_0 and r_1 at once. However, we do not implement such contract composition in the current implementation of our generative approach, although we believe that providing such support is not impossible.

Another limitation that exists in our generative approach is that a sync class must be a leaf node of a class hierarchy. To see why, consider two classes, B and D, where D directly inherits from B. Figure 5.17a illustrates the situation where B is a sync class.

Our generative approach will generate a sync controller inheriting from B. However, as D inherits from B, methods in D that it inherits from B are not safe. Therefore, D must inherit from the sync controller, as is shown in Figure 5.17b. Unfortunately, this is not the end of the story, because D might override B's methods that the controller wraps. As a consequence, calling these methods on D's instances may cause data races. Therefore, we must prevent a programmer from subclassing a sync controller. That is why our generative approach uses the final keyword to adorn every sync controller class that is generated. Hence, because a sync controller cannot be extended, the only classes that can be sync classes in a class hierarchy would be the leaf nodes (such as D), as is indicated in Figure 5.17c. This limitation may prevent our generative approach from being used in software systems having complex class hierarchies. However, we believe that this limitation can be ameliorated by more sophisticated code generation techniques. For instance, if we allow B to be a sync class, then D must also be a sync class. As a result, our approach will generate a sync controller for B and a sync controller for D (Figure 5.17d). However, generating these controllers would be complex because of issues that may arise from inheritance and polymorphism.

This chapter describes our generative approach to integrating synchronization contracts with Java. The next chapter describes our middleware approach to integrating synchronization contracts with a modern Java middleware architecture.

Chapter 6

Middleware Approach

While our generative approach shows that synchronization contracts can be elegantly integrated with Java, our middleware approach aims to investigate whether synchronization contracts can also be integrated with existing Java middleware platforms. In fact, our work on middleware synchronization contracts preceded the work on Java synchronization contracts. We initially considered using a generative approach, much like the one we used for implementing synchronization annotations. But we opted to instead introduce a new synchronization-middleware layer because this choice seemed conceptually more uniform, given that programmers are already used to programming on top of middleware. Our middleware approach also does not require the extra step of translating the application program into a synchronization-aware program.

In choosing a middleware platform, we target SIP servlets containers in the IP telecommunication (IPT) domain, because thread synchronization has proven to be difficult in this domain [37] and because the existing industry standard, JSR 289 [68], does not mandate any particular synchronization model [68, Section 6.4.2]. Here, SIP is short for Session Initiation Protocol, the de facto communication protocol in this domain; a SIP servlet is a SIP-based Java class; and a SIP servlets container is a Java middleware platform for executing SIP

servlets.

In this chapter, we first describe the architecture of SIP servlet containers (Section 6.1) and a dating service design used throughout this chapter for illustration purposes (Section 6.2). Then, we describe our middleware approach (Section 6.3) and the case study we performed to evaluate this approach (Section 6.4 and Section 6.5). Finally, in Section 6.6, we discuss some threats to validity of our case study.

6.1 Background

An IPT service is a dynamic collaboration among multiple endpoints, each of which can be thought of as a role that is played by a SIP device or by an execution of some other IPT service. An endpoint determines its player's obligations and relationships to other endpoints, much like the role played by an object determines that object's obligations and relationships to other objects in an OO collaboration [65]. For example, the roles of the endpoints in a teleconferencing service might include "administrator," whose player is responsible for maintaining the service; "leader," whose player started and is therefore responsible for ending a specific teleconference; and any number of "participants," each of whose players has joined the call. Communication with endpoints is accomplished by exchanging messages asynchronously over two-way signaling channels.

An IPT service is implemented as one or more servlets, which are deployed to a SIP servlets container [68]. The container simplifies programming by automating routine details of SIP and other generic concerns, like resource management and security. Each servlet is a stateless object to which the container dispatches incoming SIP messages for processing. Being stateless, a servlet may host multiple threads. When processing a message, a servlet stores information it will need in order to process subsequent messages in programmer-defined attributes of container-managed objects, called sessions. A container automatically

associates two types of sessions with a SIP message: an application session and a SIP session. An application session typically stores application-specific information, such as the identity of the service "administrator." Different executions of a given service often share a single application session to be able to exchange data. In contrast, they typically aggregate disjoint sets of SIP sessions. Each SIP session objectifies a signaling channel between the service execution and one of its role players (endpoints). The SIP session stores information needed to send messages to the role player, e.g., the role player's media characteristics, as well as information needed to carry out the business logic, e.g., the role played by the endpoint and the sessions needed to signal other role players. We refer to the service execution that creates a SIP session as the role player's parent execution.

To ensure responsiveness, a container executes in a dedicated thread, listening for incoming messages on the network and for outgoing messages generated by the services it hosts. Upon receiving a message bound for one of these services, the container determines the servlet to route the message to and retrieves the message's application session and SIP session, or creates them if they do not exist. It then dispatches a SIP thread to process the message, passing that thread the message and the associated servlet and sessions. The thread processes the message by invoking a message handler, which the application programmer overrides to implement the service's business logic. For instance, the application programmer typically implements the logic for processing an INVITE message in the doInvite method, the logic for processing an OK message in the doSuccessResponse method, and so on. Collectively, these methods are referred to as doXXX methods.

Smith and Bond argue that a major source of the complexity of modern telecommunication services stems from the need to maintain substantial execution state, and that use of the ECharts state-machine programming notation can alleviate much of this complexity [62]. The SIP-container API provides only low-level operations for maintaining the state of a service execution, which must be implemented using attributes in the sessions associated with

the service execution. Specifically, the API provides operations to create and destroy SIP sessions and to save and retrieve values of session attributes. It also provides operations for mapping sessions to session identifiers (IDs), suitable for transmitting over a network, and for retrieving sessions from their IDs. ECharts permits an alternate approach. The state maintenance logic is written in ECharts and then automatically translated into a SIP servlets application.

As IPT applications have evolved from single-service routing applications to applications composed of multiple services and involving multiple autonomous endpoints, the need to synchronize concurrent service executions without incurring deadlock has emerged as another significant issue for these applications. Some container implementations enforce synchronization policies automatically and also provide proprietary synchronization commands. For example, the container implemented by one popular vendor automatically locks the application session before processing any message bound to a service. A proprietary command also permits grouping sequences of operations into special transactions. Fundamental problems with these and other such mechanisms are discussed in [37].

6.2 Dating service example

We created a dating service based on a realistically complex design for use in our case study to assess the efficacy of our middleware approach (Section 6.4). To illustrate the architecture of SIP servlets containers, we now describe the design of this dating service (Section 6.2.1) and a common usage scenario (Section 6.2.2).

6.2.1 Design

Briefly, the dating service is a multi-party back-to-back user agent (B2BUA) [40], which sets up a phone call between pairs of subscribers who dial into the service to meet someone new (a

Table 6.1: Transitions in the dating service's business machine design (Figure 6.1)

label: guard -¿ actions
T1: (def Caller)?INVITE -¿ (def Msr)!INVITE(Caller)
T2: Msr?OK -¿ Caller!OK
T3: Caller?CANCEL -¿ Caller!OK; Msr!CANCEL
T4: Caller?ACK -¿ Msr!ACK
T5: Caller?BYE -¿ Caller!OK; Msr!BYE
T6: Msr?MESSAGE(def PeerToTry) && not (PeerToTry in TalkingWithMS) -¿ Msr!OK; Msr!MESSAGE(PeerToTry)
T7: Msr?MESSAGE(def PeerToTry) && (PeerToTry in TalkingWithMS) -¿ Msr!OK; def Peer = PeerToTry; Caller!INVITE(Peer); Peer to Pairing; Peer!INVITE(Caller); Msr!BYE; Peer.Msr!BYE
T8: Caller?OK -¿ Caller!ACK
T9: Msr?MESSAGE(_) -¿ Msr!OK
T10: Msr?ERROR -¿ Caller!ERROR
T11: Caller?BYE -¿ Caller!OK; Msr!BYE; Peer!BYE, Peer to Terminating

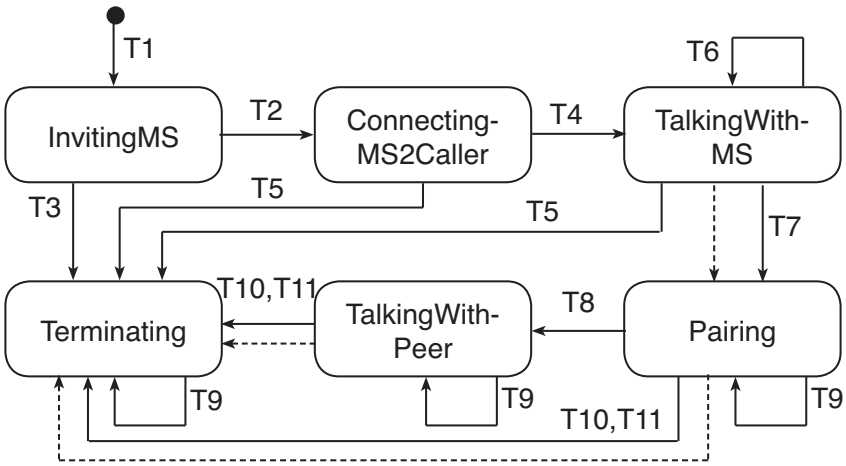


Figure 6.1: Business-machine design for a dial-up dating service

"blind date"). [1] The dating service delegates responsibility to collect match preferences from individual callers to a media service, which subsequently messages the dating service with the identities of callers who match each others' preferences. This media service is deployed as a separate IPT service.

Figure 6.1 depicts a business-machine design describing the behavior of one execution of this service. Informally, a state (rounded rectangle) keeps track of the service execution's progress between messages and a transition (solid arrow) represents the processing of a message. To reduce clutter, we depict transitions with identical source states and identical target states by annotating a single arrow with multiple labels.

Table 6.1 depicts the transitions in this business-machine. Each transition is designated by a guarded command, of the form guard -¿ action_list, which indicates the message that triggers the transition and the effects of processing this message. The guarded command notation combines ideas from CSP [34] and collaboration-based design [69]. A receive event, denoted Role?msg, models the reception of a message of type msg sent by the endpoint playing role Role. A receive event is enabled if the message to be processed is of the indicated

[1] Commercial applications of such service include omegle.com and chatroulette.com.

type and was sent by the designated endpoint's role player. Because SIP messaging is asynchronous, the sending of a message is modeled by an action rather than an event. Denoted Role!msg, the send action signifies that a message of type msg is sent to the endpoint playing role Role. Business data transmitted in a message, if any, are represented by arguments in a send action and by formal parameters in a receive event. Message transmission is asynchronous and messages need not arrive in the order sent. A role designates an endpoint, e.g., Caller refers to the phone that dialed the dating service. The qualifier def preceding a role signifies the event/action defines the role (i.e., binds it to an endpoint).

6.2.2 Usage scenario

When Mary dials the dating service, her phone sends an INVITE message to the container that hosts the dating service. Upon receiving this message, the container dispatches it to a SIP thread for processing, thereby initializing a new execution of the dating service, which we refer to as Mary's service execution. The initial transition (T1) represents the processing of this message. It binds the Caller role in Mary's service execution to Mary's phone (the endpoint that sent the message). It then creates and sends a new INVITE message to a media service, binding the media-service execution created by the media service when it receives this message to the Msr role in Mary's service execution. This latter message carries an identifier designating Mary's phone (the caller), which the media service will need if Mary matches another caller's preferences. When the SIP thread processing Mary's original INVITE message terminates, Mary's service execution is in state InvitingMS. Because the SIP sessions for the endpoints that play the Caller and Msr roles are created in processing this message, Mary's service execution is the parent service execution for both endpoints.

If the media-service execution subsequently responds with an OK message, Mary's dating-service execution relays this response to her phone (T2) and transitions into state ConnectingMS2Caller. A subsequent ACK response from Mary's phone is similarly relayed to the

media service (T4). This exchange completes a three-way handshake between Mary's phone and the media service, permitting Mary's phone and the media service to exchange media directly with one another. While Mary's phone and the media service are connected, Mary's service execution is in state TalkingWithMS.

Normally, Mary's service execution waits in this latter state until it either receives a message containing the identity of a caller that matches Mary's preferences or Mary is selected by some other caller. Either of these events requires that Mary's service execution performs a transition synchronously with another caller's service execution. For instance, suppose Mary signals a preference for Tom, the caller in another dating-service execution, while interacting with her own media-service execution. Her media-service execution will then notify her dating-service execution of her selection. However, by the time Mary's service execution receives this notification message, Tom may no longer be available (e.g., Tom may have hung up in the interim, or he may have selected yet a third caller to be connected with). Thus, to ensure that neither Tom's nor Mary's service execution get into an inconsistent state, the reception of the notification message should trigger a simultaneous state change in both Mary's and Tom's service executions, but only if both are still available to be paired.

This rendezvous style of synchronization can be implemented by acquiring locks on all sessions used to record the execution state of any peer in the rendezvous, and modifying the attributes of these sessions in a single critical section. In a business machine, we show these rendezvous operations more abstractly using guard conditions and actions that refer to the states of multiple service executions and using special transitions with no guarded commands. Specifically, the guard condition (role in state) queries the state of the parent service execution of the endpoint designated by role, returning true if the service execution is in state state and false otherwise; and the action (role to state) transitions this same service execution to state state. A transition with no guarded command (dashed arrow) signifies

that when a service execution is in the transition's source state and not processing any message, another service execution may perform an action that leaves it in the transition's target state.

For instance, suppose Mary's service execution receives a MESSAGE message while it is still in state TalkingWithMS and the identifier passed in for PeerToTry designates the endpoint that plays the caller in Tom's service execution. If Tom's service execution is also in state TalkingWithMS, Mary's service execution transitions both her own and Tom's service executions into state Pairing (T7). In this state, Mary's service execution attempts to connect Mary's phone and Tom's phone. On the other hand, if Tom's service execution is not in state TalkingWithMS when Mary's service execution receives the MESSAGE message, Tom is no longer available, and Mary's service execution arranges for Mary to select a different caller (T6). At any time, Mary can terminate her service execution by hanging up (T3, T5, or T11). Additionally, while the dating service is trying to connect the media service to Mary's phone, the media service can signal an error has occurred (T10).

6.3 SNeF4SS: A synchronization middleware

Because the doXXX methods in a SIP servlet are not traceable to the design of a business-machine that a programmer creates, we invented a new message handler doTransition to help improve such traceability (Section 6.3.3). As a result, the programmer implements the actions that are performed on transitions; and a synchronization contract specifies the conditions under which a servlet performs transitions and the sessions that it accesses when performing them (Section 6.3.2). By consulting the contract and the message handler, a synchronization service can automatically emulate the business machine while preventing data races and deadlock (Section 6.3.4).

We have developed a reference framework, called Synchronization Negotiation Frame-

work for SIP Servlets (SNeF4SS)[2] [36]. SNeF4SS is a middleware layer between a servlet and the container. To facilitate the description of SNeF4SS, we begin by describing the pseudocode conventions used throughout the remainder of the chapter (Section 6.3.1).

6.3.1 Pseudocode conventions

Prior to writing the contracts or any message handlers for a servlet, the programmer must decide how she will represent concepts in the business machine as data objects that a message handler can manipulate when processing a message. Because a servlet is stateless, the handler must use the servlets-container API to navigate from the message-processing context to any data it needs. Thus, the data (or references to the data) are typically stored as values of attributes in SIP sessions.

For the dating service, we give every SIP session an attribute, named role, that is assigned one of two values: "Caller", if the SIP session objectifies the service execution's channel to its caller; or "Msr", if it objectifies the service execution's channel to the media service. Before processing a message, the handler checks the value of this attribute to determine which endpoint sent the message. The other attributes in a SIP session depend on the value of the role attribute:

- if role == "Caller":

  cs: the state of the service execution

  msr: the SIP session for the service execution's channel to the media service

  p2t: the SIP session in a peer's service execution for the channel to the peer execution's caller

- if role == "Msr":

[2] http://www.cse.msu.edu/sens/szumo/SNeF

nav_expr	=	mess_desc
	—	sess_desc, { "." , name }
mess_desc	=	"msg", ".", ("method" — "payl")
sess_desc	=	"msg", ".", ("apps" — "sips")
	—	name

Figure 6.2: Navigation expressions. (Extended BNF notation: "typewriter" – terminal; italics – non-terminal; comma – concatenation; vertical bar – alternation; set brackets – (optional) repetition; equal – definition; parentheses – grouping.)

caller: the SIP session for the service execution's channel

to its caller

We use this attribute scheme and a simple version of navigation expressions [24] in describing both contracts and the processing performed in message handlers. A navigation expression is a "dot-separated" sequence of two or more names (Fig. 6.2). It consists either of a message descriptor or of a session descriptor, where the latter may be followed by any number of names. A message descriptor stands for information that is found in the message. We use two in this example: msg.payl stands for the information transmitted in the payload of a message, and msg.method stands for the method named, in the first line of a SIP request message, or the type of response designated by the response code, in the first line of a SIP response message. Thus, for example, when processing a response message, the navigation expression msg.method signifies the response type (e.g., OK). In contrast to a message descriptor, a session descriptor specifies how to locate session data. The session descriptors msg.apps and msg.sips indicate that the traversal starts in the message's application session and in the message's SIP session, respectively. Any other name used as a session descriptor denotes a variable that references a session. Additional names attached to a session descriptor designate a series of attributes to be traversed to reach a data object.

For example, when processing a message sent by a service execution's caller, the navigation expression msg.sips.msr stands for the SIP session for the service execution's channel to the media service.

Finally, we introduce three abstract functions to use in pseudocode descriptions.

- send(method, payl, chan): Creates and sends a message. Arguments indicate the method/type of the response code, the payload, and the channel (recipient).
- getID(ss): Returns the session identifier that the container associates with a session.
- getSession(ssid): Returns the session indicated by a session ID.

6.3.2 Synchronization contracts

SNeF4SS synchronization contracts declare guard conditions for the transitions in a business machine and handles for the sessions that the transitions manipulate. These declarations permit the middleware to automatically acquire sessions, as they are needed, in order to emulate execution of the business machine without incurring data races or deadlock.

SNeF4SS reads an XML encoding of a table that defines a servlet's contracts. Table 6.2 depicts some of the contracts for the dating service example. Each row of this table names a transition (first column), specifies a guard expression (middle column) and a set of handles (last column), and defines any abbreviations used in the contract (spanning the last two columns). The guard condition specifies when the transition is executed as a boolean expression. The handles specify the set of data objects to which a message handler assumes it has exclusive access for the duration of a transition—namely, all sessions that are traversed in evaluating any of the handles. For example, the first row of Table 6.2 specifies that an OK response, if sent by the media service and received while a service execution is in state InvitingMS, triggers transition T2, and that during execution of this transition, the handler

Table 6.2: Synchronization contracts for transitions triggered by OK and MESSAGE messages.

Name	Guard	Handles
T2	msg.method == OK && msg.sips.role == "Msr" && msg.sips.caller.cs == "InvitingMS"	msg.sips.caller
T6	p2t = getSession(ssid = msg.payl)	
	msg.method == MESSAGE && msg.sips.role == "Msr" && msg.sips.caller.cs == "TalkingWithMS" && (p2t == null —— p2t.cs != "TalkingWithMS")	msg.sips.caller, p2t
T7	p2t = getSession(ssid = msg.payl)	
	msg.method == MESSAGE && msg.sips.role == "Msr" && msg.sips.caller.cs == "TalkingWithMS" && p2t != null && p2t.cs == "TalkingWithMS"	msg.sips.caller, p2t.msr
T8	msg.method == OK && msg.sips.role == "Caller" && msg.sips.cs == "Pairing"	msg.sips
T9	msg.method == MESSAGE && msg.sips.role == "Msr" && (msg.sips.caller.cs == "Pairing" —— msg.sips.caller.cs == "TalkingWithPeer" —— msg.sips.caller.cs == "Terminating")	msg.sips

assumes it has exclusive access to the SIP session for the caller (msg.sips.caller) and, because the SIP session for the media service (msg.sips) is traversed in evaluating the handle, to this latter SIP session as well. The table illustrates contracts for the transitions triggered by two types of SIP messages: OK messages and MESSAGE messages. The former is representative of a response type and the latter is the request type requiring the most complex processing in this example. SNeF4SS automatically translates navigation expressions into appropriate calls on the container API when evaluating contracts.

6.3.3 The message handler

When writing a servlet that will run directly on the container API, a programmer typically implements the actions described by a business machine in multiple message-specific doXXX methods. For a servlet that will use the SNeF4SS API, the programmer instead overrides a single abstract handler method, called doTransition (Fig. 6.3).

The doTransition method is passed the name of the transition to emulate and the message to process. Based on the transition name, it selects a branch of a conditional statement, which carries out the actions for that transition. For example, the first branch of the conditional (lines 2–9) in Figure 6.3 implements the actions performed by the initial transition, T1, of the business machine in Figure 6.1. Briefly, it initializes attributes role and cs in the message's SIP session (lines 3 and 4, resp.); creates a new SIP session to serve as the service execution's channel to the media service and assigns this new session to attribute msr (line 5); initializes attributes caller and role in this new session (lines 6 and 7, resp.); and sends an INVITE message to the media service (line 8) containing an identifier, which the container associates with the caller's SIP session.

When refining a business machine into an actual servlet, a programmer substitutes calls to the container API for the pseudocode shown in our figures. This step is routine, but tedious and prone to error because of the low level at which the programming must currently be

```
1  void doTransition (name:String, msg:SIPMessage) {
     if (name == "T1") { caller = msg.sips
3      caller.role = "Caller"
       caller.cs = "InvitingMS"
5      caller.msr = new SIP session
       caller.msr.caller = caller
7      caller.msr.role = "Msr"
       send(method = INVITE, payl = getID(ss = caller),
9           chan = caller.msr) }
     else if (name == "T2") { msr = msg.sips
11     send(method = OK, chan = msr.caller)
       msr.caller.cs = "ConnectingMS2Caller" }
13   ...
     else if (name == "T6") { msr = msg.sips
15     send(method = OK, chan = msr)
       send(method = MESSAGE, payl = msg.payl, chan = msr) }
17   else if (name == "T7") { msr = msg.sips
       caller = msr.caller
19     p2t = getSession(ssid = msg.payl)
       send(method = OK,  chan = msr)
21     caller.p2t = p2t
       caller.cs = "Pairing"
23     p2t.cs = "Pairing"
       // connect caller and p2t
25     send(method = INVITE, chan = caller, payl = p2t.SDPinfo)
       send(method = INVITE, chan = p2t, payl = caller.SDPinfo)
27     // tear down both channels to  media service
       send(method = BYE, chan = msr)
29     send(method = BYE, chan = p2t.msr) }
     else if (name == "T8") {caller = msg.sips
31     send(message = ACK, chan = caller)
       caller.cs = "TalkingWithPeer" }
33   else if (name == "T9") {
       send(message = OK, chan = msg.sips) }
35   ...
   }
```

Figure 6.3: Pseudocode for part of a handler to use with SNeF4SS

done. Understanding the actual servlet code requires a deeper understanding of SIP than can be conveyed here. However, the actual source code is available for download at our website.[3] Moreover, a conclusion of our case study is that this translation could be largely automated.

6.3.4 Operation of the framework

When a SNeF4SS servlet is deployed to a container, the servlet's contract is loaded and parsed into an internal representation for later evaluation relative to different message-processing contexts. Subsequently, upon being dispatched, a SNeF4SS thread evaluates this internal representation relative to the message that it was dispatched to process, producing a dynamic contract. The dynamic contract contains the name of the transition, if any, that the message triggers, together with references to the sessions obtained by evaluating the handles associated with this transition. If the dynamic contract is null, the message does not trigger any transition, and so the SNeF4SS thread just terminates without calling the handler. This helps to automatically absorb resent or irrelevant messages. More typically, the dynamic contract is non-null and the thread then performs a negotiation protocol to acquire the needed sessions. During negotiation the thread may yield to other threads contending for some of the same sessions in order to prevent deadlock. When it succeeds in acquiring the needed sessions, a SNeF4SS thread invokes the servlet's doTransition method, passing it the name of the transition and the original message. Finally, when the call to doTransition returns, the thread releases all sessions and terminates.

In essence, SNeF4SS encapsulates a distributed concurrency controller[8]. It automatically creates an isolated zone [20, Chap. 5] containing the resources needed by the handler. There are many protocols for acquiring a set of resources without incurring deadlock [72]. Known protocols range in complexity, from relatively simple algorithms [25], which may be

[3] http://www.cse.msu.edu/sens/szumo/SNeF/

overly pessimistic or unfair, to complex algorithms [6, 18], which can permit more concurrency and be fairer, but at some extra costs in overhead. To separate the implementation of the negotiation protocol from the rest of the framework, SNeF4SS encapsulates the protocol implementation in negotiator objects, which implement a generic Negotiator API. The current SNeF4SS toolkit provides three alternative negotiators for an application programmer to use off-the-shelf. The three negotiators each implement a different negotiation protocol.

Encapsulating the synchronization code within a framework in this manner promises a number of benefits. First, it frees the application programmer to focus on design and implementation of the business logic. As illustrated in Figure 6.3, the servlet code does not tangle business code with synchronization code. The business logic is thus relatively easy to trace to the servlet code. Second, experts in concurrent programming can write the implementation of the locking protocol to be efficient under expected usage profiles and validate that it avoids deadlock and starvation. The application programmer, who is an expert in the business domain, but may not be proficient in thread programming, does not write the code to acquire and release sessions. Use of the framework can thus offer strong guarantees with regard to efficiency and absence of concurrency errors. Third, alternative synchronization protocol implementations can be seamlessly swapped into the framework to accommodate changes in a service's usage profile. Concurrency experts can develop custom negotiators by programming to the SNeF4SS Negotiator API [38].

6.4 Case study

Having introduced the business-machine design for the dating service and described our framework, we now turn to describing the case study [39]. We developed and compared three implementations of the design described in Section 6.2 in an attempt to understand tradeoffs between design transparency and performance. The goal was to determine if ad-

ditional investment in generative programming of IPT services from high-level designs and synchronization contracts is warranted. Toward this end, the case study looked at two key questions:

1. How does servlet code written to run directly on the container API compare to servlet code written to run on SNeF4SS with respect to design transparency and code complexity?

2. What are the performance costs to use contract-based synchronization instead of hand-written synchronization?

This section considers Question 1, which is difficult to quantify, but is important for reasoning about correctness and determining how well the framework would support automated generation of IPT services. The next section considers Question 2, which is a concern because quality-of-service is a critical differentiator among providers in this domain [16].

We originally intended to re-engineer an existing service to use as the subject of the case study. However, we abandoned this plan for two reasons. First, the field of IPT services has grown out of the telecommunications domain, in which software tends to be proprietary. The only open source services that we found were those packaged with open source containers to show how to use the container API; they were far too simple to serve as subjects for our case study. We are currently working with the SPEC SIP Committee on development of a benchmark for evaluating SIP application servers,[4] but this effort started long after we performed the case study reported here. A second reason for developing a service from scratch was to be able to address the two questions identified. For Question 1, we needed to start with a well-engineered business-machine design. For Question 2, we needed to control carefully for both differences in the implementations and for factors not related to synchronization that could affect our measurements. However, mindful of this

[4] http://www.spec.org/specsip

potential threat to validity, we provide the full source code for all versions of the dating service developed for this study and instructions for replicating our results.[5]

For the case study, we implemented the business machine from Section 6.2 three times. Two of the implementations target the container API and the third targets the SNeF4SS API. We refer to the first two as manual versions, because they embed custom code to synchronize concurrent executions, and to the third as the contract version, because it leverages synchronization contracts. We then compared the manual versions to the contract version on measures that speak to the two questions.

To facilitate comparison and highlight tradeoffs between design transparency and efficiency in the three implementations, we first describe how the business logic is expressed in servlet code, without concern for synchronization (Section 6.4.1). Then we describe how the different approaches for implementing the synchronization logic affect this servlet code (Section 6.4.2).

6.4.1 Implementing the business logic

Before writing any code, we designed a scheme for storing and retrieving persistent data in SIP sessions. The requirements for such a scheme are similar, whether targeting the container API or the SNeF4SS API. When processing a message, the handler must find any data it needs by navigating the message-processing context. The scheme that we started with for all versions is that described in Section 6.3.1.

Manual versions. The manual versions must conform to the programming rules in JSR 289 [68]. According to this standard, the servlet inherits from the class SipServlet and overrides the appropriate doXXX methods. When a SIP thread is dispatched to process a message, it invokes the servlet's service method. This latter method examines the message type and then invokes the appropriate doXXX method. A SIP thread performs these oper-

[5] http://www.cse.msu.edu/sens/szumo/SNeF

ations automatically. Therefore, the application programmer need only write code for the doXXX methods to implement the business logic.

Because the doXXX method that a SIP thread invokes depends on the type of the message to be processed, we partitioned the transitions of the business machine, putting all transitions that are triggered by a given message type into the same equivalence class. We then implemented the transitions for an equivalence class as a doXXX method.

To illustrate, Figure 6.4 shows pseudocode for two representative handlers, doSuccessResponse and doMessage. Briefly, when the dating service receives an OK response message, it checks whether the response was sent by the caller (line 2) or the media service (line 7). It then checks the current state. If the caller sent the response and the service execution is in state Pairing (line 2), the handler implements transition T8. If the media service sent the response and the service execution is in state InvitingMS (line 3), the handler implements transition T2. In all other cases, the message has no affect on the service execution. The most complex handler is doMessage (lines 14–31). It does not check the role of the sender because the caller does not send MESSAGE messages to the dating service. However, if the service execution is in state TalkingWithMS, the handler checks if the service execution of the prospective peer is also in state TalkingWithMS, implementing transition T7, if it is, and transition T6, otherwise (lines 22–24 and lines 19–21, resp.). Finally, the handler executes T9 (a self loop) in all other cases (lines 26–30).

Contract version. For the contract version, instead of inheriting directly from SipServlet, the servlet class inherits from a base class that SNeF4SS defines via subclassing SipServlet. The new base class overrides the service method from the SipServlet class. The SNeF4SS service method does not delegate handling of different messages to different handlers; instead, it uses a negotiator object to learn the transition to emulate and then invokes doTransition, passing both the transition name and the message as arguments. Like a SIP thread, a SNeF4SS thread automatically invokes the servlet's service method. Therefore, the appli-

```
void doSuccessResponse (msg: SIP Message) {
  if (msg.sips.role == "Caller") {
    if (msg.sips.cs == "Pairing") {
      // T8 (Fig. 6.3, lines 30--32)
    }
  }
  else if (msg.sips.role == "Msr") {
    if (msg.sips.caller.cs == "InvitingMS") {
      // T2 (Fig. 6.3, lines 10--12)
    }
  }
}

void doMessage(msg: SIP Message) {
  msr = msg.sips
  caller = msr.caller
  p2t = getSession(ssid = msg.payl)
  if (caller.cs == "TalkingWithMS") {
    if (p2t == null or p2t.cs != "TalkingWithMS") {
      // T6 (Fig. 6.3, lines 14--16)
    }
    else {
      // T7 (Fig. 6.3, lines 17--29)
    }
  }
  else {
    // caller.cs is one of: "Pairing" or "TalkingWithPeer"
    // or "Terminating"
    // T9 (Fig. 6.3, line 34)
  }
}
```

Figure 6.4: Pseudocode for two SIP handlers

cation programmer's job consists of writing code for the doTransition method to implement the actions performed on transitions of the business machine. Section 6.3.3 describes this process and shows pseudocode for handling OK and MESSAGE messages (Fig. 6.3).

Discussion. The example pseudocode in Figures 6.3 and 6.4 illustrates one advantage of a middleware that leverages contracts to support the business machine abstraction. Declaring transitions and associated guard conditions in separate contracts that the middleware uses to automatically determine the transition to emulate permits the use of a handler whose structure very transparently parallels the structure of the business machine. The application programmer does not need to partition the business logic into handlers for different types of SIP messages, or clutter the handler with code to check the role of the message sender or the execution state. The use of explicit transition names in the SNeF4SS handler makes it self-documenting. Tracing transitions of the business machine to the code in message-specific SIP handlers can be difficult, especially in the absence of comments spelling out the correspondence.

6.4.2 Implementing the synchronization logic

Threads hosted by the same container must be properly synchronized to protect shared sessions from data races and to prevent deadlock. All three versions of the dating service use the same locking protocol, resource numbering [29], to avoid deadlock. In order to better understand tradeoffs between design transparency, code complexity, and efficiency, one of the manual versions uses the same generic ordering scheme as the contract version—ordering sessions by hash codes that the Sun Java Virtual Machine (JVM) assigns to objects—and the other employs an application-specific ordering scheme.

First manual version. For the first manual version, we tried to retain as much design transparency as possible. Toward this end, we extended the base servlet class with two new methods: acquire( ) aggregates the logic for each message type to infer and acquire locks on

all needed sessions; and release( ) releases all locks held by the handler. We then added a call to acquire as the first instruction and a call to release as the last instruction in each doXXX method.[6]

The application programmer must write the acquire( ) and release( ) methods. It is easy to write a "generic" release( ) method, which simply releases the locks on all sessions held by the message handler. However, the acquire( ) must infer the sessions that should be locked, which requires precisely the information encoded in our contracts. In the absence of explicit declarations encoding this information, the application programmer must write an acquire( ) method that hard codes this information.

To infer the sessions that are needed, the acquire( ) method in our first manual version aggregates the control logic of all handlers: First, it determines the message type, the role played by the endpoint that sent the message, and the state of the service execution. Based on this information, it determines the actions that must be performed and the sessions that will be accessed in performing these actions. During this inference process, the method locks a session when necessary to traverse an attribute in that session (e.g., to infer a session some action will access). Because the traversal order may be inconsistent with the hash-code order used to prevent deadlock, the code to evaluate a navigation expression that traverses multiple sessions without introducing deadlock is complex. It must allow that, to acquire a lock on a new session, the handler may have to release locks on sessions that are already held and reacquire them once it has locked the new session, and that the states of the reacquired sessions may change between the time they are released and the time they are reacquired. To avoid this complexity and extra overhead, in this first manual version of the dating service, we added a cs attribute to the SIP session for each service execution's channel to the media service; this attribute simply duplicates the cs attribute in the SIP

[6] with one exception, doINVITE—In theory, a race on the message's SIP session could occur between doINVITE and doCANCEL, but the likelihood is extremely small and, in any case, the race would be benign because doCANCEL invalidates the signaling channel with the caller.

session for the service execution's channel to the caller. By duplicating this information in both SIP sessions of a service execution, we arrange that only the SIP session associated with the message is needed in order to determine all sessions that must be acquired.[7] Thus, our acquire( ) method locks the message's SIP session and infers the sessions needed to process the message. It then proceeds to acquire the needed sessions in hash-code order, releasing and reacquiring the message's SIP session if necessary to acquire all the sessions in hash-code order. It is unlikely that, in the time between releasing the message's SIP session and reacquiring it, the cs attribute will have changed. However, if it does, the method can simply restart.

Duplicating the cs attribute in the SIP sessions for both role players in a service execution simplifies the synchronization logic in the acquire( ) method, but requires adding to the servlet of the previous subsection code that keeps the duplicate attribute values consistent. It also means that, at a minimum, a thread must acquire the sessions for both of the role players in a service execution when processing any message. For instance, suppose a thread is dispatched with an OK message sent by the caller in a service execution when the service execution is in state Pairing (transition T8). In the first manual version the thread must acquire both the message's SIP session and the SIP session for the media service in order to update the cs attributes in both sessions; whereas, in the optimized and contract versions, only the message's SIP session is needed.

Second manual version. The second manual version trades design transparency in order to optimize performance. In this version, we embedded synchronization code directly into the branches of handlers, thereby eliminating the need to duplicate the control logic by which a handler infers the actions to perform in the acquire method. Doing the acquisition and release in the same method also allows use of Java synchronized statements instead of

[7] for T7, the msr attribute of the peer session can be safely accessed without locking the peer session because the attribute is a "single-assignment" attribute

explicit locks.

To simplify the locking logic, we developed a custom order for acquiring resources. We first ordered the SIP sessions in any given service execution so that the SIP session for the caller immediately precedes that for the media service. Then we used the hash-code order between the SIP sessions for the callers in different service executions to induce a total order on the full space of SIP sessions. Finally, as a further optimization in this version, we leveraged the knowledge that attribute role is a single assignment attribute—i.e., role is assigned a value at initialization and is never reassigned—to reduce the size of critical regions.

Figure 6.5 illustrates the structure of the optimized handlers for OK and MESSAGE messages. Integrating the synchronization logic into doSuccessResponse is straightforward. No locks are needed when determining the role that the message sender plays (lines 2 and 8). If it plays the caller role, the handler needs to lock only the message's SIP session (line 4). Otherwise, the handler may need both of the service execution's SIP sessions. The prescribed acquisition order (a.o.) requires the handler to first acquire the SIP session for the caller (line 10) and then, if needed, the SIP session for the media service (line 12).

In contrast, integrating the synchronization logic into the doMessage method makes the method significantly more complex. The handler must implement logic for two possible acquisition orders. We show the logic for just one of them (lines 26-38). The elided branch (line 39) is similar in complexity to the one shown, but is not a mirror image because the acquisition order affects the order of the nested conditionals.

Contract Version. For the contract version of the dating service, we wrote contracts declaring guard conditions and handles for the 11 transitions in Figure 6.1. The contracts are written in XML. They all follow the pattern of those illustrated in Table 6.2.

Discussion. Targeting the SNeF4SS API reduced both the amount and the complexity of the code that an application programmer needs to write for this example. The lines of code

```
1  void doSuccessResponse(msg:SIPMessage) {
     if (msg.sips.role == "Caller") {
3      // a.o.: msg.sips < msg.sips.msr
       synchronized (msg.sips) {
5        if (msg.sips.cs == "Pairing") {
           // T8 (Fig. 6.3, lines 30--32)
7    } } }
     else if (msg.sips.role == "Msr") {
9      // a.o.: msg.sips.caller < msg.sips
       synchronized (msg.sips.caller) {
11       if (msg.sips.caller.cs == "InvitingMS") {
           synchronized (msg.sips) {
13           // T2 (Fig. 6.3, lines 10--12)
   } } } } }
15
   void doMessage(msg:SIPMessage) {
17   // (Fig. 6.4, lines 15 - 17)
     synchronized (msr) { // factor out Msr!OK
19     // T9 (Fig. 6.3, line 34)
     }
21   if (p2t == null) {
       synchronized (msr) {
23       // T6' (Fig. 6.3, lines 14, 16)
         return
25   } }
     if (caller < p2t) {
27     // a.o.: caller < msr < p2t < p2t.msr
       synchronized (caller) {
29       if (caller.cs == "TalkingWithMS") {
           synchronized (msr) {
31           synchronized (p2t) {
               if (p2t.cs != "TalkingWithMS") {
33               // T6' (Fig. 6.3, lines 14, 16)
                 }
35               else {
                   synchronized (p2t.msr) {
37                   // T7' (Fig. 6.3, lines 17-19, 21-29)
     } } } } } } }
39   else { // a.o.: p2t < p2t.msr < caller < msr }
   }
```

Figure 6.5: Portion of an optimized handler

(LOC) for the message handlers in the optimized version (the shorter of the two manual versions) total to approximately 350. The message handler in the contract version contains approximately 200 LOC. Counting each contract as 5 LOC brings an estimate of LOC for the contract version to about 250.

The additional code in the manual versions accounts for most of the complexity of the code that the application programmer must write. As an indication of the complexity of the synchronization code in the optimized version, we tabulated the number of synchronized statements and their synchronized-nesting depths. Six of the message handlers in this version contain synchronized statements. Altogether, these 6 handlers contain 19 synchronized statements: eight at depth 1, five at depth 2, four at depth 3, and two at depth 4. The synchronization code dwarfs the business code in doMessage and doBye, which contain the depth-3 and depth-4 synchronized statements.

Synchronization code, whether encapsulated in a framework or written by the programmer, must infer the sessions that need to be acquired. Thus, the programmer must identify both guard conditions and handles, whether using the container API or the SNeF4SS API. When using SNeF4SS, the programmer expresses this information declaratively in a contract and leaves the framework to determine when and in what order to lock sessions. When using the container API, the programmer writes low-level operational code that infers the sessions needed and then acquires them in a consistent order. This code is significantly more complex than the code implementing the business logic. If mixed into the handlers, it obscures the business code and makes tracing the business design to the code very difficult.

6.5 Performance evaluation

The previous section presents the three implementations developed for the feasibility study and demonstrates that contract-based synchronization can provide benefits for design and

development. This section describes the empirical evaluation we performed to assess whether dynamically interpreting contracts and using a generic negotiation protocol to acquire the needed sessions is practical in this domain.

Briefly, we ran each of the three implementations of the dating service in the same test environment and under essentially the same loads and usage profiles. For each run, we calculated two widely-used performance metrics for SIP containers—throughput and response time. We then compared how these metrics scale as the call rate increases. The remainder of this section provides details of the test environment (Sec. 6.5.1) and the test runs (Sec. 6.5.2), and then presents and discusses the performance results (Sec. 6.5.3 and 6.5.4, resp.).

6.5.1 Test environment

Our test environment contains five computers on a 100M switched ethernet network: two servers, one running the dating service and the other running the media service; two load generators, each simulating sets of callers; and one test coordinator, which reboots the servers prior to each run, triggers the load generators, and collects performance data.

Each server has a 64-bit Intel Xeon Quad Core 1.6GHz processor and 4G of memory. On each, we run 64-bit Linux (2.6 kernel), 64-bit JVM (Sun JDK 1.6.0 Update 14), and Sailfin SIP Servlet Container (1.0).[8] To reduce the impact of garbage collection, we configure the JVM using the guidelines in [64].

Each load generator has a 32-bit Intel Pentium IV 2.4GHz CPU and 1G of memory. The load generators run 32-bit Linux (2.6 kernel) and SIPp (3.1),[9] a widely used SIP load generating tool, to simulate concurrent callers. The hardware and software for the test coordinator are not material to the performance evaluation.

[8] http://sailfin.dev.java.net

[9] http://sipp.sourceforge.net

This hardware and software determines the capacity of the test environment. In test runs of the optimized version of the dating service, we observed that at about 100 concurrent service executions, the container starts to thrash. A call rate is unsustainable when messages from uncompleted service executions start to accumulate. We therefore regard 100 concurrent service executions as representing an overload for our configuration of the dating service. In contrast, our tests indicate that the optimized version can sustain 80 concurrent service executions indefinitely. We therefore regard 80 concurrent service executions as the expected load for our test configuration.

6.5.2 Test runs

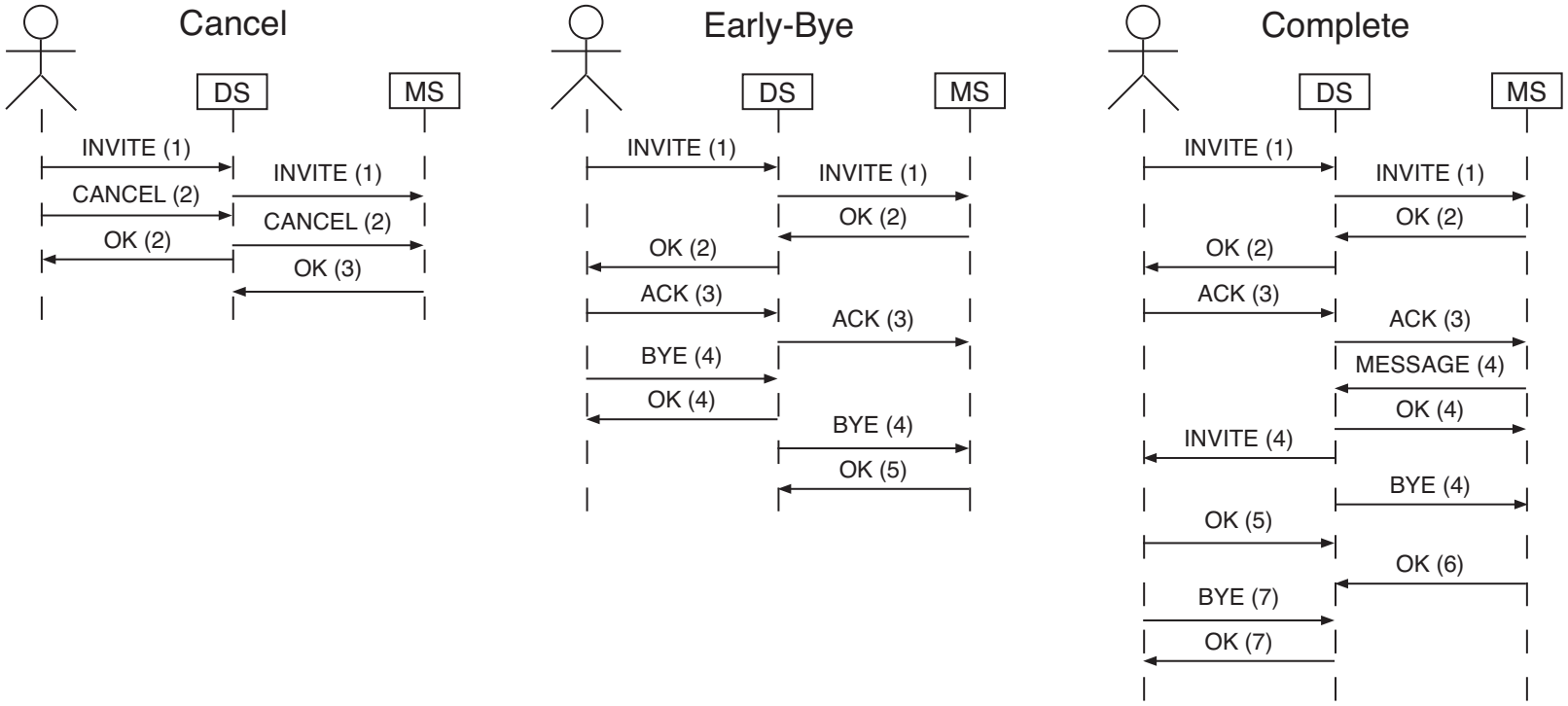


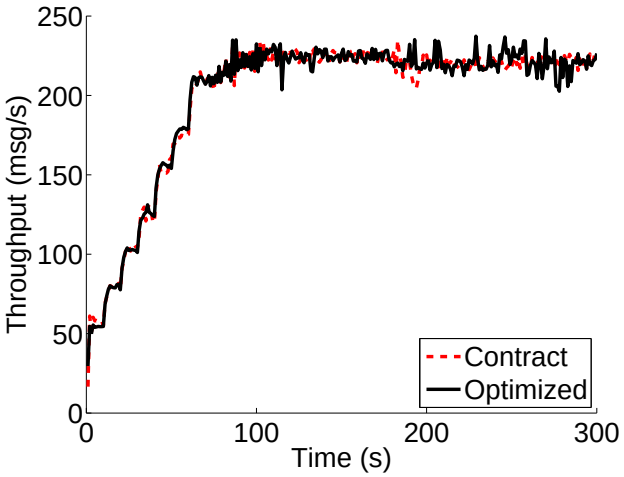
Figure 6.6: Sample message flows produced by three representative usage profiles

A test run simulates a set of callers over a period of 300 seconds. Because performance is affected both by a caller's usage profile and by the call rate, a test run controls for both. We distinguish three usage profiles, which produce different message flows (Figure 6.6): Cancel (left) simulates a caller who hangs up immediately after placing the call; Early Bye (middle) simulates a caller who hangs up after her phone acknowledges the first OK response and before being paired with another caller; and Complete Call (right) simulates a caller who talks with another caller for two seconds and then hangs up. To indicate messages that are processed and sent by the same thread, we show thread number in parentheses. In practice, we expect callers to exhibit a mixture of these usage profiles, with the majority executing Complete Call. For a test run, therefore, we fixed the probabilities of the usage profiles at 5% Cancel, 5% Early Bye, and 90% Complete Call, randomly assigning a usage profile to each simulated caller with these probabilities.

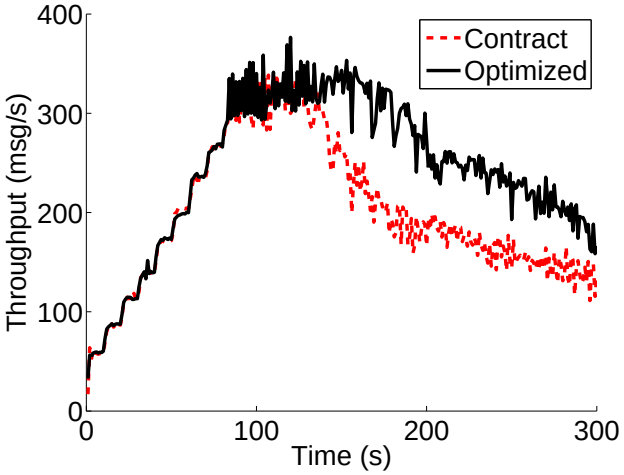
To measure performance at expected load, we started each test run at 10 calls-per-second (cps) and increased the call rate by 5 cps every 10 seconds. Upon reaching expected load, or 80 concurrent service executions, we adjusted the call rate to keep the load just under 80. To measure differences in scalability, we increased the call rate by 5 cps every 10 seconds until 100 concurrent service executions was reached, instead of 80. We then adjusted the call rate to keep the number of concurrent service executions near 100. Each test starts with a warmup phase, during which containers are booted and data structures are initialized. No data is collected during this phase, as it is not material to performance of the service. We ran each test 10 times, each time for 300 seconds. The numbers reported are thus averages of 10 runs over these 300 seconds.

6.5.3 Performance results

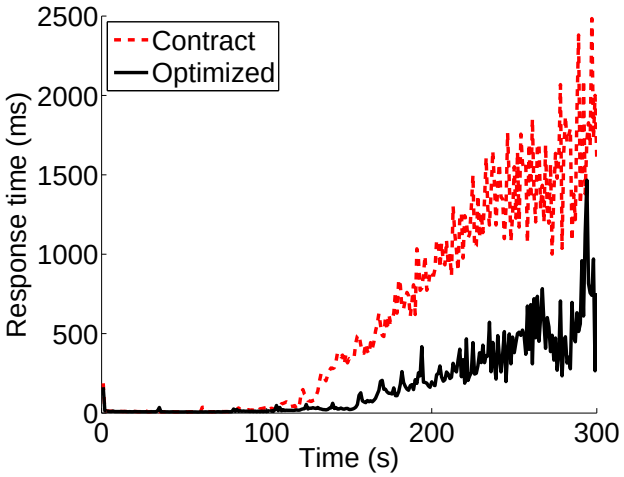
The first manual version and the contract version of the dating service performed essentially the same in our test runs. For this reason and for lack of space, we present results of just the



(a)



(b)



(c)

Figure 6.7: Performance comparison between optimized and contract versions (left: throughput at expected load; middle: throughput when overloaded; right: response time when overloaded)

optimized and contract versions. Figure 6.7 plots results of measuring throughput (left two graphs) and response times[10] (right graph) for the optimized version (black solid) and the contract version (red dashed). We report throughput while operating under expected load (left) and under an overload (middle).

For throughput, we analyzed the system logs generated by the container, reporting the average number of messages processed per second during a run. Both implementations exhibited essentially the same throughput in the first 120 seconds. Prior to the 90th second, the throughput increased in a "step-like" fashion because the call rate increased by 5 every 10 seconds. Throughput leveled off as the number of concurrent service executions neared their targets: 80, which occurred at about the 70th second, under expected load; and 100, which occurred at about the 90th second, under the overload. The callers then started to maintain the target loads. In the case of expected load, throughput is essentially identical for the full 300 seconds. In the case of overload, both implementations continue to exhibit similar throughput for approximately another 30 seconds. After this period, the throughput of the contract implementation begins to drop when overloaded, whereas that of the manual implementation remains relatively stable until the 160th second, at which time it also begins to drop.

For response time, we averaged all per-caller response times reported by SIPp, where a per-caller response time is the time elapsed between when the caller sends the INVITE message invoking a dating-service execution and receives an OK response from the dating service. As in the case of message throughput, we observed no noticeable difference in the performance of the optimized and contract versions as long as the system was not overloaded. But after about 30 seconds at near 100 concurrent service executions, we began to observe better measures for the optimized version. For the contract version, the response time is under 20 ms for 93% of the calls, and under 50 ms for 97%. For the optimized version, these

[10] also referred to as call-setup delay

percentages are 98% and 99%, respectively. Although the optimized version established more calls (5%) within 20 ms, both versions established most calls (97% and 99%) within 50 ms.

6.5.4 Discussion: Performance

Because the first manual version and the contract version use the same generic locking protocol, we expected them to exhibit the same scaling behavior. We attribute the finding that the actual measures observed for the contract version were as good as those for the first manual version to the fact that the first manual version needs to spend extra time in evaluating the guards in the acquire( ) method. Essentially, the overhead to evaluate the contract is similar to that to evaluate the guards.

The optimized version and the contract version exhibit the same scaling behavior when running near expected load. However, they exhibit different scaling behavior from the 120th second to the 160th second, when the container is overloaded. We believe this difference stems from two main sources. First, the contract version employs explicit locks to synchronize threads, whereas the manual version uses the Java synchronized statement. Locks provide the flexibility needed to separate the synchronization code from the business code, but incur some extra costs. Specifically, SNeF4SS creates a dedicated lock for each SIP session and maintains an internal mapping between a SIP session and its lock. Second, the implementation of resource numbering that we used for the contract version is fully generic in inferring the sessions needed and the locking order from the contract and a total order defined by the JVM; whereas the optimized version leverages an application-specific order relation and knowledge that certain attributes are assigned only once.

Of these two sources of differences, we believe that the latter is more significant and, moreover, that performance of a contract-based version can get even closer to that of the optimized version by adding information about single-assignment attributes to contracts. When processing a message from the caller, a negotiator first locks the caller's session; if

the negotiator then finds that it also needs to lock the session for the media service and if the media service's session precedes the caller's session in hash-code order, it rolls back, unlocking the caller's session, then rolls forward, locking first the media service's session and then the caller's session, and finally rechecks the the guard condition in case it became stale during the back-off. In a dating-service execution, we expect approximately half of the messages bound for the service execution to be from the caller and about half from the the media service. Moreover, the handler needs both the caller's session and the media service's session to process most messages. Thus, a negotiator frequently back-offs, even if no other negotiator is contending for a session it needs. Each back-off incurs extra overhead. The optimized version avoids this problem by checking single-assignment attributes outside of any critical regions. Thus, in future work, we intend to add tags to our contract language for SNeF4SS that declare single-assignment attributes and add another generic negotiator implementation to our current toolkit. The new negotiator will leverage single-assignment declarations to avoid acquiring locks unnecessarily.

6.6 Threats to validity

We are aware of several threats to validity of our case study. First, the dating service is but one data point. While we designed it to be realistic and representative of the kinds of IPT services deployed today, we may have failed to capture some critical aspect of their complexity. Also, we focused our performance analysis on specific usage profiles and in specific proportions. An earlier analysis, however, using just Call Complete, the most complex profile, produced similar results. Second, we chose to use a locking protocol based on a specific policy, resource numbering. The comparison might differ if the manual implementation used a more ad hoc policy. That said, it would have been difficult to fairly compare the contract-based synchronization to manual synchronization if each used a fundamentally

different locking protocol. Finally, it is possible that, despite our best efforts, some bias toward our approach entered into the implementations of the different versions of the dating service developed for the study. We address this threat by making the source code public[11], thereby allowing other researchers to replicate our results.

[11] http://www.cse.msu.edu/sens/szumo/SNeF

Chapter 7

Conclusion

The complexity of implementing thread synchronization in multi-threaded programs stems from having to interleave the synchronization code with the functional code. This thesis describes a synchronization approach that permits a programmer to develop safe, flexible, and efficient multi-threaded programs without manually implementing the synchronization code. The key of this synchronization approach is the use of declarative synchronization contracts for specifying the synchronization logic on a fairly high-level.

We investigated in two specific approaches to integrating synchronization contracts. The generative approach allows synchronization contracts to be annotated in the source code; it then leverages a Java compiler plugin to automatically weave the synchronization code, which is translated from the contracts, into the synchronization-agonistic class that a programmer writes. We believe that the annotation model that our generative approach adopts presents an elegant model for specifying a program's synchronization logic. Specifically, this model not only allows a programmer to directly think about the synchronization needs for executing a functional method in terms of resources and conditions but also allows a programmer to explicitly document such synchronization needs using synchronization annotations. Such manner of specifying the synchronization logic gives several important software engineering

benefits. For one thing, it facilitates the development of multi-thread programs, because synchronization annotations are easier to write, read, and debug than low-level synchronization code. For another, it improves a program's readability, not only because synchronization annotations are easier to read but also because the functional code is easier to read without synchronization code embedded with it. Therefore, we believe that investment in leveraging synchronization annotations and the Java compiler to automate thread synchronization for multi-threaded programs can be very promising.

The other integration approach that we investigated targets an existing Java middleware platform in the IPT domain. This approach allows synchronization contracts to be specified separately in an IPT service's deployment descriptor; the approach then leverages an independent synchronization middleware to automatically load and enforce the contracts as the service processes messages. We believe our success in the middleware approach owes to several characteristics of IPT services. First, all of the sessions needed to process a message are reachable from the message-processing context, and navigation expressions provide a concise and compositional means for specifying how to dynamically infer these sets of sessions. Second, the short—under 0.2 ms [10]—bursty nature of message processing is conducive to synchronizing by acquiring all of the needed sessions before performing any actions on them and releasing them all when all actions have been performed. Third, the information that a programmer needs to know to craft a synchronization contract is mostly available in the state machine that describes the business logic. By enabling more concurrency, a synchronization service based on incremental resource acquisition should, in theory, improve performance. However, any such approach must employ protocols for avoiding or recovering from deadlock, and such protocols incur additional overhead during contract negotiation. While such overhead may be reasonable in environments where request processing may take a significant amount of time, such case is not true for IPT services, where servlet activations typically process a message in a very short amount of time (typically 1–2 milliseconds) [10].

For synchronization contracts to be truly useful in practice, however there is yet a lot of work to be done. For example, the negotiator API needs to be general enough to permit expression of diverse synchronization requirements; it also needs to be standardized to ensure portability. However, designing and drafting such standard would require participation of many language companies, such as Oracle and Microsoft. Additionally, the generated synchronization code must be highly optimized, which requires efforts from not only the concurrency experts but also the compiler optimization community. Therefore, we hope that the pilot work and the initial results described in this thesis will be used to justify future investment in contract-based synchronization by practitioners, researchers, and engineers in related communities.

Chapter 8

Future Work

In future work, we will investigate how well synchronization annotations can be applied in realistically complex Java systems. For example, it would be interesting to know whether synchronization annotations can replace the synchronization code of an existing Java system without major changes to the functional code; whether use of synchronization annotations will facilitate creation of a more robust design; whether use of synchronization annotations will force a programmer to come up with a design that deviates from what she would normally devise; and how "thinking in synchronization annotations" will affect a programmer's mental load in designing or developing a system—in other words, can time or effort can be saved by using synchronization annotations in the tasks of debugging or maintaining software? Toward this end, we recently started a project, in which we are re-engineering the synchronization code in Android mobile operating system and replacing its synchronization code with synchronization annotations; we are also collaborating with colleagues at Oracle in looking for internal development teams, who will be interested in applying our generative approach in developing new software modules.

Another interesting direction for future work is to create a semantic foundation for using synchronization annotations, which is necessary for formal proof and verification. In describ-

ing our generative approach, we give a list of programming rules of using synchronization annotations. We claim that a program that is created by following these rules should exhibit no data races. However, lacking a formal semantics of synchronization annotations, we are not able to give a formal proof of this claim. A semantic foundation for synchronization annotations would allow us to prove that this list of rules is complete and sound. Furthermore, a semantic foundation would also help us identify additional rules that might provide other synchronization guarantees, such as absence of deadlocks or absence of livelocks.

Another promising avenue for future research is automated generation of synchronization annotations. Currently, writing correct and optimal synchronization annotations requires that a programmer follows the programming rules to ensure that the synchronization contracts are satisfied. However, a programmer could easily forget to enforce a rule when developing a program; for example, she might forget to add a sync resource to the resources property of a sync method's annotation when she extends the sync method's body with a statement that uses that sync resource. In this case, the synchronization contracts of this program will become unsatisfied and, consequently, the program may exhibit data races. Although this case might be detected by a formal verification process, a more ideal resolution would be to prevent this case from happening in the first place by automatically generating the synchronization annotations. For example, the sync resources that a sync method (or a sync condition) uses might be obtained by parsing the body of that sync method (or that sync condition); the guard conditions and the impact conditions of a sync method might be automatically inferred from a program's invariants. For instance, consider a sync class that implements a bounded buffer. If the invariant for this class is $0 <= buf.size() <= MAX$, then this invariant will tell us that the add(...) method should not make the buffer exceed the buffer's maximum size MAX; as a consequence, the guard condition of this method would be $buf.size() < MAX$. Similarly, the guard condition of the remove(...) method would be $buf.size() > 0$. Likewise, the impact conditions of these methods could also be

automatically inferred from this invariant and the bodies of these methods.

Regarding our middleware approach, future work might explore the substantial benefits that synchronization contracts provide for compositional reasoning about IPT services based on the business-machine designs [13]. For example, given some number of clients who invoke a service, we might model the overall behavior of the service as a process in Finite State Processes (FSP) [51] by (1) writing a process B representing the business machine, (2) for each client, creating the client process $c{:}B$, in which actions of B are prefixed with a unique client name c, (3) identifying shared transitions,[1] and (4) forming the parallel composition of the client processes under a renaming that forces shared transitions to occur simultaneously. Using the Labeled Transition System Analyzer (LTSA) tool [51] we could then interactively simulate scenarios of interest, formally verify or refute temporal properties, and generate error traces. For instance, we might verify that the dating service never connects two different subscribers to the same peer and that a subscriber cannot be in state Pairing if the selected peer is in state Terminating. Another use of composition in reasoning about services is in modeling a service composed of multiple sub-services. We could obtain a model of the composition by composing appropriately specialized models of the sub-services, and then analyze the composite model to check for errors in the overall business logic. For instance, to check if a subscriber who phones the dating service will eventually be connected to the media server provided she does not hang up first, we might analyze an FSP process obtained by composing a process representing some number of concurrent dating-service executions with a FSP process representing the same number of concurrent media-service executions (produced from a business machine describing the media service's business logic) under an appropriate renaming to capture the messaging semantics. To reduce potential for state explosion, we could hide actions in the sub-processes not pertinent to the property or to

[1] e.g., the dashed transition in $c{:}B$ from c.TalkingWithMS to c.Pairing[c'] and the corresponding solid transition in $c'{:}B$ from c'.TalkingWithMS to c'.Pairing[c]

interactions between the dating-service and media-service executions and minimize the sub-processes prior composing them.

A potential pitfall in the analysis scenario outlined above is that the validity of such analysis depends on transitions being atomic, but the processing of messages by a service is not atomic. Thus, to ensure the analysis results hold for an actual service, we need to know, not just that the message handlers correctly update the execution state, but also that each behavior of the service is equivalent to a behavior in which messages are processed serially. The contract-based method for developing a servlet facilitates showing that the message handlers correctly implement the business logic by producing methods that do not mix synchronization code with business code and by using a conditional statement whose conditions explicitly identify the names of the transitions effected by the branches. In addition, if a servlet satisfies its synchronization contract, invoking the doTransition function creates an isolated zone [20], consisting of all resources a thread might access in processing the message—thereby guaranteeing that the message-processing operations in a servlet that uses synchronization contracts are serializable. Moreover, in the IPT domain, it should be possible to determine the handles a thread might use when processing a message by inspecting the code, since the thread locates each resource it accesses using the servlet-container API to navigate from the message-processing context to the resource.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. E. Allchin and M. S. McKendry. Synchronization and recovery of actions. SIGOPS Oper. Syst. Rev., 19:32–45, January 1985.

[2] S. I. Anthony and T. A. Marsland. The deadlock problem: An overview. IEEE Computer, 13:58–78, 1980.

[3] J. Armstrong and T. Helen. Making reliable distributed systems in the presence of software errors, 2003.

[4] S. Balzer, P. Eugster, and B. Meyer. Can aspects implement contracts? In N. Guelfi and A. Savidis, editors, Rapid Integration of Software Engineering Techniques, volume 3943 of Lecture Notes in Computer Science, pages 145–157. Springer Berlin / Heidelberg, 2006.

[5] R. Behrends. Designing and Implementing a Model of Synchronization Contracts in Object-Oriented Languages. PhD thesis, Michigan State University, East Lansing, Michigan USA, Dec. 2003.

[6] R. Behrends and R. E. K. Stirewalt. The Universe Model: an approach for improving the modularity and reliability of concurrent programs. In Proc. FSE'00: ACM SIGSOFT 8th Intl. Symp. Foundations of Softw. Eng., pages 20–29, San Diego, CA, USA, November 2000. ACM.

[7] R. Behrends, R. E. K. Stirewalt, and L. K. Dillon. A self-organizing component model for the design of safe multi-threaded applications. In Proc. of the ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE'05), 2005.

[8] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In Proc. ASE '04: 19th IEEE Intl. Conf. Automated Softw. Eng., pages 248–257, Lawrence, KA, USA, September 2004. IEEE Computer Society.

[9] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? Queue, 6(5):46–58, 2008.

[10] M. Cortes, J. Ensor, and J. Esteban. On SIP performance. Bell Labs Technical J., 9(3):155–172, Jan 2004.

[11] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In ASPLOS-XII: Proc. of the 12th international conference on Architectural support for programming languages and operating systems, pages 336–346, New York, NY, USA, 2006. ACM.

[12] X. Deng et al. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In Proc. of the IEEE International Conference on Software Engineering (ICSE'02), 2002.

[13] L. K. Dillon, Y. Huang, and R. Stirewalt. Leveraging synchronization contracts for compositional reasoning in design of services. In FLACOS 10: Fourth Workshop on Formal Languages and Analysis of Contract-Oriented Software, 2010.

[14] L. K. Dillon, R. E. K. Stirewalt, B. Sarna-Starosta, and S. D. Fleming. Developing an Alloy framework akin to OO frameworks. In Proc. of the First Alloy Workshop, 2006. co-located with FSE'2006.

[15] A. Dinning. A survey of synchronization methods for parallel computers. Computer, 22:66–77, July 1989.

[16] D. Ferrari. Client requirements for real-time communication services. IEEE Communications, 28(11):65–69, 1990.

[17] F. Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional synchronization primitives. In Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, PODC '04, pages 80–87, New York, NY, USA, 2004. ACM.

[18] C. M. Fleiner and M. Philippsen. Fair multi-branch locking of several locks. In Proc. ICPDCS'97: Intl. Conf. Parallel and Distributed Computing Systems, pages 537–545, Washington, DC, USA, October 1997.

[19] M. Fowler. Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[20] M. Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

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

[22] G. Granunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. Computer, 23:60–69, June 1990.

[23] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci., 410:202–220, February 2009.

[24] A. Hamie, J. Howse, and S. Kent. Navigation expressions in object-oriented modelling. In E. Astesiano, editor, Proc. FASE'98: Fundamental Approaches to Softw. Eng., volume 1382 of Lecture Notes in Computer Science, pages 123–138. Springer Berlin / Heidelberg, 1998.

[25] P. B. Hansen. Operating System Principles. Prentice Hall, USA, 1973.

[26] T. Harris and K. Fraser. Language support for lightweight transactions. In Proc. of the ACM SIGPLAN Conference on Object-Oriented Systems, Languages, and Applications (OOPSLA'2003), 2003.

[27] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[28] M. Haustein and K.-P. Löhr. JAC: declarative Java concurrency: Research articles. Concurr. Comput. : Pract. Exper., 18(5):519–546, 2006.

[29] J. W. Havender. Avoiding deadlock in multitasking systems. IBM Systems J., 7(2):74–84, 1968.

[30] C. T. Haynes and D. P. Friedman. Engines build process abstractions. In Proceedings of the 1984 ACM Symposium on LISP and functional programming, LFP '84, pages 18–24, New York, NY, USA, 1984. ACM.

[31] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA '06, pages 253–262, New York, NY, USA, 2006. ACM.

[32] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In PODC '03: Proc. of the twenty-second annual symposium on Principles of distributed computing, pages 92–101, New York, NY, USA, 2003. ACM.

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

[34] C. A. R. Hoare. Communicating sequential processes. Commun. ACM, 21(8):666–677, 1978.

[35] C. A. R. Hoare and R. H. Perrott Ed. Towards a theory of parallel programming. London: Academic, 1972.

[36] Y. Huang, E. Cheung, L. K. Dillon, and R. E. K. Stirewalt. A thread synchronization model for SIP servlet containers. In Proc. IPTComm '09: 3rd Intl. Conf. Principles, Systems and Applications of IP Telecommunications, pages 1–12, Atlanta, GA, USA, 2009. ACM.

[37] Y. Huang, L. K. Dillon, and R. E. Stirewalt. On mechanisms for deadlock avoidance in SIP servlet containers. In Proc. IPTComm '08: 2nd Intl. Conf. Principles, Systems and Applications of IP Telecommunications, pages 196–216, Heidelberg, September 2008. Springer-Verlag.

[38] Y. Huang, L. K. Dillon, and R. E. K. Stirewalt. Prototyping synchronization policies for existing programs. In Proc. ICPC'09: Intl. Conf. Program Comprehension, pages 289–290. IEEE Computer Society, 2009.

[39] Y. Huang, L. K. Dillon, and R. E. K. Stirewalt. Contract-based synchronization of IP telecommunication services: A case study. In The Fifth International Conference on COMmunication System softWAre and middlewaRE. ACM, 2011.

[40] A. B. Johnston. SIP: Understanding the Session Initiation Protocol. Artech House, Inc., Norwood, MA, USA, 3rd edition, 2009.

[41] G. Kiczales et al. Aspect oriented programming. In European Conference on Object-Oriented Programming (ECOOP'97), 1997.

[42] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In J. Knudsen, editor, ECOOP 2001 Object-Oriented Programming, volume 2072 of Lecture Notes in Computer Science, pages 327–354. Springer Berlin / Heidelberg, 2001.

[43] J. Kienzle and R. Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In B. Magnusson, editor, ECOOP 2002 Object-Oriented Programming, volume 2374 of Lecture Notes in Computer Science, pages 113–121. Springer Berlin / Heidelberg, 2006.

[44] M. F. Kilian. A conditional critical region pre-processor for c based on the owicki and gries scheme. SIGPLAN Not., 20:42–56, April 1985.

[45] F. Klein, A. Baldassin, J. Moreira, P. Centoducatte, S. Rigo, and R. Azevedo. Stm versus lock-based systems: an energy consumption perspective. In Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design, ISLPED '10, pages 431–436, New York, NY, USA, 2010. ACM.

[46] J. Larus and C. Kozyrakis. Transactional memory. Commun. ACM, 51(7):80–88, 2008.

[47] D. Lea. Concurrent Programming in Java. Addison–Wesley, second edition, 2000.

[48] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. Commun. ACM, 44:39–41, October 2001.

[49] A. Lister. The problem of nested monitor calls. SIGOPS Oper. Syst. Rev., 11:5–7, July 1977.

[50] V. Luchangco and V. J. Marathe. Transaction synchronizers. In 2005 Workshop on Synchronization and Concurrency in Object Oriented Languages (SCOOL'05), Oct. 2005. held at OOPSLA'05.

[51] J. Magee and J. Kramer. Concurrency: State Models and Java Programs. John Wiley and Sons, second edition, 2006.

[52] R. C. Martin. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[53] R. C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008.

[54] T. Mens and M. Wermelinger. Separation of concerns for software evolution. Journal of Software Maintenance, 14(5):311–315, 2002.

[55] B. Meyer. Applying design by contract. IEEE Computer, 25(10), 1992.

[56] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In Proceedings of the 1st international conference on Aspect-oriented software development, AOSD '02, pages 141–147, New York, NY, USA, 2002. ACM.

[57] B. Ramesh, D. Mahata, and S. Sharma. Event-based synchronization, October 2006. Patent no. US 7,117,496 B1.

[58] M. C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. ACM Trans. Comput. Syst., 17:337–371, November 1999.

[59] S. Rugaber, K. Stirewalt, and L. M. Wills. The interleaving problem in program understanding. Reverse Engineering, Working Conference on, 0:166, 1995.

[60] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison–Wesley, second edition, 2004.

[61] B. Sarna-Starosta, R. E. K. Stirewalt, and L. K. Dillon. A model-based design-for-verification approach to checking for deadlock in multi-threaded applications. In Proc. of 18th Intl. Conf. on Softw. Eng. and Knowledge Eng., 2006.

[62] T. M. Smith and G. W. Bond. ECharts for SIP servlets: a state-machine programming environment for voip applications. In Proc. IPTComm '07: 1st Intl. Conf. Principles, Systems and Applications of IP Telecommunications, pages 89–98, New York, NY, USA, July 2007. ACM.

[63] R. E. K. Stirewalt, R. Behrends, and L. K. Dillon. Safe and reliable use of concurrency in multi-threaded shared memory sytems. In Proc. of the 29^{th} Annual IEEE/NASA Software Engineering Workshop, 2005.

[64] B. Van Den Bossche, F. De Turck, B. Dhoedt, P. Demeester, G. Maas, J. Moreels, B. Van Vlerken, and T. Pollet. J2EE-based middleware for low latency service enabling platforms. In Proc. GLOBECOM '06: Global Telecommunications Conf., pages 1–6, San Francisco, CA, USA, November 2006.

[65] M. VanHilst and D. Notkin. Using role components in implement collaboration-based designs. In Proc. OOPSLA '96: 11th ACM SIGPLAN Conf. Object-oriented programming, systems, languages, and applications, pages 359–369, San Jose, CA, USA, October 1996. ACM.

[66] M. Wand. Continuation-based multiprocessing. Higher Order Symbol. Comput., 12:285–299, October 1999.

[67] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, and P. Schulthess. Optimistic synchronization and transactional consistency. In Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID '02, pages 331–, Washington, DC, USA, 2002. IEEE Computer Society.

[68] J. Wilkiewicz and M. Kulkarni. JSR 289: SIP servlet specification v1.1. http://jcp.org/aboutJava/communityprocess/final/jsr289.

[69] R. Wirfs-Brock and A. McKean. Object Design: Roles, Responsibilities and Collaborations. Addison–Wesley, 2003.

[70] C. Zhang. Flexsync: An aspect-oriented approach to java synchronization. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 375–385, Washington, DC, USA, 2009. IEEE Computer Society.

[71] C. Zhang and H.-A. Jacobsen. Externalizing java server concurrency with cal. In J. Vitek, editor, ECOOP 2008 Object-Oriented Programming, volume 5142 of Lecture Notes in Computer Science, pages 362–386. Springer Berlin / Heidelberg, 2008.

[72] D. Zöbel. The deadlock problem: a classifying bibliography. SIGOPS Oper. Syst. Rev., 17(4):6–15, 1983.