UNIVERSITY OF CALIFORNIA
Santa Barbara

Design for Verification for Concurrent and Distributed Programs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

by

Aysu Betin-Can

Committee in Charge:

Professor Tevfik Bultan, Chair

Professor Chandra Krintz

Professor Peter Cappello

December 2005

The dissertation of Aysu Betin-Can is approved.

Professor Peter Cappello

Professor Chandra Krintz

Professor Tevfik Bultan, Committee Chair

September 2005

Design for Verification for Concurrent and Distributed Programs

Aysu Betin-Can

To my wonderful and dearest husband,

Tolga Can

and

To my family *Ayten, Suat, Cansu Betin*

Acknowledgements

I will remember my years in pursuing PhD degree as a gratifying and engaging experience thanks to many people including my research advisor, the faculty and the staff in my department, my friends, my parents, my sister, and my husband.

First of all, I would like to thank Professor Tevfik Bultan for being a great advisor throughout my graduate study. His enthusiasm, great ideas and directions helped me stay on the right track all the time. His trust in me encouraged me to overcome the hardest problems. I was inspired by his extensive knowledge, wisdom, and motivation. I am proud of being one of his students.

I would also like to thank Professor Chandra Krintz and Professor Peter Cappello for serving on my committee and for keeping their doors open all the time for any of the questions I had.

I have really enjoyed to work with my fellow students and friends. I would like to thank Tuba Yavuz-Kahveci, Xiang Fu, Constantinos Bartzis, Graham Hughes, and Çağdas Gerede for the fruitful discussions we had. I would also like to thank the Computer Science staff for all their efforts to help us achieve our goals. They made us feel as part of the family.

I would like to thank my parents and my sister for supporting me and believing in me all through my life as a student. Their support through phone calls from overseas has been really valuable to me for the last five years. Last, but not the least, I would like to thank my wonderful husband Tolga for always being with me with his endless love and support.

Curriculum Vitæ

Aysu Betin-Can

Education

September 2005	Doctor of Philosophy in Computer Science, University of California, Santa Barbara.
June 1999	Bachelor of Science in Computer Engineering, Middle East Technical University, Ankara, Turkey.

Fields of Study

Software engineering, reliable software development, modular verifi cation and specifi cation, web services, concurrency, model checking.

Publications

Aysu Betin-Can, Tevfi k Bultan, Mikael Lindvall, Stefan Topp and Benjamin Lux. “Application of Design for Verifi cation with Concurrency Controllers to Air Traffi c Control Software”. In the Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE 2005), Long Beach, California, USA, November 7-11, 2005.

Tevfi k Bultan and Aysu Betin-Can. “Scalable Software Model Checking Using Design for Verifi cation.” To appear in the Proceedings of the IFIP Working Conference on Verifi ed Software: Theories, Tools, Experiments, Zurich, Switzerland, October 10-14, 2005.

Aysu Betin-Can and Tevfi k Bultan. “Verifi able Web Services with Hierarchical Interfaces”. In the Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005), pp. 85-94, Orlando, Florida, USA, July 11-15, 2005.

Aysu Betin-Can, Tevfi k Bultan and Xiang Fu.“Design for Verifi cation for Asynchronously Communicating Web Services”. In Proceedings of the 14th International World Wide Web Conference (WWW 2005), pp. 750-759, Chiba, Japan, May 10-14, 2005.

Aysu Betin-Can and Tevfik Bultan. "Verifiable Concurrent Programming Using Concurrency Controllers." In the Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004), pp. 248-257, Linz, Austria, September 20-25, 2004.

Aysu Betin-Can. "Design for Verification for Concurrent and Distributed Systems", SIGSOFT 2004 Student Research Forum in conjunction with the SIGSOFT 12th International Conference on Foundations of Software Engineering, Newport Beach, California, November 2004.

Aysu Betin-Can and Tevfik Bultan. "Interface-Based Specification and Verification of Concurrency Controllers." In Proceedings of the Workshop on Software Model Checking (SoftMC 2003) in conjunction with the 15th International Conference on Computer Aided Verification, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 89, no. 3. Boulder, Colorado, July 14, 2003.

Aysu Betin-Can and Tevfik Bultan. "Interface-Based Specification and Verification of Concurrency Controllers". Technical Report No: 2003-13 in Department of Computer Science University of California Santa Barbara

Abstract

Design for Verifi cation for Concurrent and Distributed Programs

by

Aysu Betin-Can

In this dissertation we present a design for verification (DFV) approach that embeds intentions of developers into software and makes software systems amenable to automated verification; hence, making the automated verification techniques scalable to large systems. In this DFV approach, we use 1) behavioral interfaces that isolate the behavior and enable modular verification, 2) an assume-guarantee style verification strategy that separates verification of the behavior from the verification of the conformance to the interface specifications, 3) a general model checking technique for interface verification, and 4) domain specific and specialized verification techniques for behavior verification.

We realize our DFV approach for concurrent programming by introducing the concurrency controller pattern. We aim to eliminate synchronization errors in concurrent Java programs. We use the Action Language Verifier to verify the concurrency controller behaviors by an automated translation from their Java implementations. We have applied this framework to two software systems: a concurrent text editor and a safety critical air traffic control software called TSAFE.

To demonstrate the applicability of our DFV approach to another application domain, we introduce the peer controller pattern for asynchronously communicating

web services. Our goal is both to analyze properties of interactions among the participating peers and to validate the conformance of peer implementations to their behavioral specifications. We use the SPIN model checker to verify the interaction properties. We adapt synchronizability analysis to enable behavior verification with respect to unbounded asynchronous communication queues. We extend this approach with an hierarchical interface model for compact representation of peer interfaces.

We use the Java PathFinder for interface verification in both application domains. We present techniques for thread isolation which improve the efficiency of interface verification.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Assuring reliability and correctness for complex software systems is a major challenge. Correctness and reliability are the most important software qualities especially for safety critical systems. For example, recently, a collaborative research program, called NASA High Dependability Computing Program (HDCP) [54], was established to investigate ways to improve the development of highly reliable software. There are various automated verification tools and techniques for assuring the reliability and correctness of a software system. Some of the examples are Java PathFinder [99], SPIN [58], ESC/Java [43], and Action Language Verifier [23]. In all of these tools, however, scalability is an issue.

Model extraction is the most crucial step for scalable software verification. A compact model hides the details that are irrelevant to the properties being verified. Such models are extracted through user guidance [53], static analysis techniques [5, 55], or both [25]. Model extraction rediscovers some information about the software that may be known to its developers at design time. A design for verification approach that enables software developers to document the design decisions and

makes verifiability an explicit design criterion [75], can be useful for verification and may improve scalability. We summarize this approach as follows:

> Designing software systems in a way that includes the developer's intentions and makes the system amenable to automated verification; hence, making the automated verification techniques scalable to large systems. We call this approach *Design for Verification* (DFV).

In our design for verification methodology, we achieve this goal by using domain specific behavioral design patterns. Any software system developed based on the patterns introduced in this dissertation embodies the developer's intentions and the model of the software system explicitly. The resulting software is suitable for automated verification, improving the applicability of model checking techniques significantly.

The DFV approach introduced in this dissertation is based on the following principles: 1) use of stateful, behavioral interfaces which isolate the behavior and enable modular verification, 2) an assume-guarantee style verification strategy which separates verification of the behavior from the verification of the conformance to the interface specifications, 3) a general model checking technique for interface verification, and 4) domain specific and specialized verification techniques for behavior verification. We have applied our DFV approach based on these principles to verification of synchronization operations in concurrent programs [11, 13, 16] and to verification of interactions among multiple peers in composite web services [14, 15]. We have developed two verifiable design patterns which facilitate modular specification of interfaces and behaviors. These verifiable design patterns also help us

in automating the model extraction and environment generation steps required for software model checking.

Our assume-guarantee style modular verification strategy is based on the stateful interfaces that isolate the behavior. An interface of a component should provide the necessary information about how to interact with that component without giving all the details of its internal structure. However, providing only names, types, and signatures, as commonly practiced in interface representations in current programming languages, is not sufficient for most verification tasks [21]. For example, such an interface does not contain any information about the order in which the methods of the class should be called. Therefore, we use finite state machines to specify interfaces. Such interface machines can be used to specify the order of method calls or any other information that is necessary to interact with a component. For complex interfaces one could use an extended state machine model and provide information about some of the input and output parameters of the component. In Chapter 3, we discuss interfaces in detail and introduce a formal model for such interfaces. Another possible extension is to use hierarchical state machines for interfaces [14] which is discussed in Section 5.5.

The verification strategy we present in this dissertation exploits the decoupling of behavior and interface specifications in the proposed design patterns and combines the strengths of different model checkers. Some model checkers have their own specification languages [23, 58, 72]. The usage of such model checkers in software verification requires abstractions of the input programs in order to translate them to the input languages of these model checkers [25]. These model checkers are spe-

cialized for different aspects of verification and employ specialized model checking techniques on the specifications written in their own input languages. These techniques, however, cannot handle all the constructs in a high level programming language. Model checkers that target a programming language [99, 49, 43], on the other hand, can handle all the constructs in a high level language. The weakness of these more generalized techniques, however, is that they cannot explore the whole state space due to state space explosion. Our techniques combine these two approaches as follows:

1. During behavior verification, we use domain specific model checkers. To verify the synchronization behavior in concurrent programs we use symbolic model checking techniques. These techniques enable us to verify the behavior in the presence of parameterized constants, unbounded variables, and arbitrary number of concurrent threads. To verify the interactions among multiple peers in an asynchronously communicating web services, we use the model checking techniques that focus on process interaction and can handle asynchronous communication among processes.

2. Conformance to the interface specifications is checked by using model checkers that target a programming language. We use Java PathFinder [99] which is a model checker for Java programs and can handle arbitrary Java implementations without any restrictions. Conformance is verified using interfaces as stubs for controllers and isolating concurrent threads and asynchronous peers. This approach improves the efficiency of the interface verification significantly.

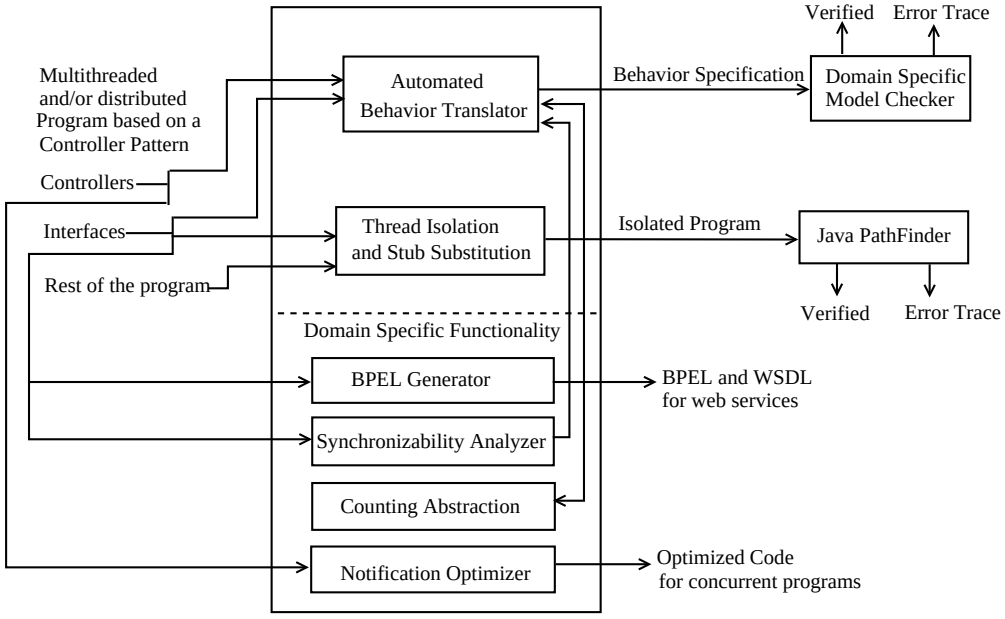


Figure 1.1: System architecture overview

The proposed modularization of the verification tasks improves the efficiency of the verification, covers a larger portion of the state space, and hence, makes automated verification of realistic programs feasible.

An overview of our approach based on the proposed verifiable design patterns is illustrated in Figure 1.1. A program written based on the these design patterns consists of controllers, their interfaces, and the rest of the program. These components are shown on the left side of the figure. The assume-guarantee style verification approach is illustrated on the top half of the figure. The verification consists of two steps: behavior and interface verification. During behavior verification, a controller and its interface are given to our automated behavior translator. (The behavior of each controller is verified separately.) The output of this translator is a behavioral

specification of the program written in the language of a domain specific model checker. This specification is verified with respect to user defined properties. The result of this phase is either a certification that the behavioral properties hold, or falsification with an error trace. For the interface verification, the input is the controller interfaces and the rest of the program. Interface verification is performed on each concurrent thread in isolation. During the isolation process the interactions among the application threads are abstracted by stub substitution and replacement of controllers with their interfaces. The result of this isolation is a set of isolated programs, one per application thread. In the case of asynchronously communicating web services, the same process is used to isolate the participant peers with peer interfaces. Each of these isolated programs is checked for interface violations with a model checker for Java. The result of interface verification is either an assurance that the users of a controller obey the interfaces, or an error trace showing the violation in the implementation with respect to the controller interface.

In addition to the above verification approach, our framework also includes domain specific functionalities shown at the bottom of the main box separated with a dashed line in Figure 1.1. For composite web service systems, we provide an automated BPEL generator that synthesizes BPEL and WSDL specifications from the interfaces to be published for interoperability. This functionality exploits the explicit definitions of interfaces in the peer controller pattern. We also provide a synchronizability analyzer in order to be able to verify properties of asynchronously communicating composite web services in the presence of unbounded message queues. A composite web service is called synchronizable if its global behavior does not

change when asynchronous communication is replaced with synchronous communication [47]. The analyzer checks the sufficient conditions for synchronizability presented in [47] based on the peer controller pattern using the peer interfaces and works with the automated behavior translator to generate a behavior specification with synchronous communication semantics. For concurrent programs, we provide a notification optimizer that eliminates unnecessary context switches among concurrent threads. This optimizer performs an automated notification analysis on the concurrency controllers. The output is a new concurrency controller implementation that uses the specific notification pattern [24] and has the same behavior of the original concurrency controller. The last feature, called counting abstraction, works with the automated behavior translator. The counting abstraction enables us to verify the behavior of a concurrency controller for arbitrary number of application threads.

Design for verification is a fairly new concept. Sharygina et al. [90] focus on verification of UML models, whereas we focus on verification of programs. Similar to our work, Mehlitz et al. [75] also suggest using design patterns in improving the efficiency of the automated verification. Our interface-based modular verification technique, however, is different and does not overlap with their approach [75]. Design patterns are utilized by Mehlitz et al. for code separation, to partition a large program into independently verifiable components. They suggest having usage rules for these patterns; however, the specification method is not precisely defined since the work is in its early stages. The design decisions are reflected in the code by source code annotations and dynamic assertions [74]. These annotations are suggested to be used for extracting models from the code. Design for verification methodology

has also been applied to circuit design and hardware systems [89, 66, 3].

There has been other work on behavioral and stateful interfaces. Chakrabarti et al. [27] specify interfaces of software modules as a set of constraints, and present algorithms for interface compatibility checking. The authors have applied their interface formalism also to web service interfaces [18, 17]. We use finite state machines to specify interfaces and, unlike their work, our goal is to verify both the controller behavior and conformance to interface specifications. DeLine et al. [32, 33] extend type systems with stateful interfaces and treat interface checking as a part of type checking. There has also been work on interface discovery and synthesis in which stateful interfaces are extracted by analyzing existing code [100, 2]. In contrast, we use interfaces as part of a verifiable design pattern to isolate the synchronization behavior or asynchronous interaction behavior.

Summary of Contributions

The contributions in this dissertation can be summarized as follows.

• We introduce a DFV approach for concurrent programs with a concurrency controller pattern and for asynchronously communicating web services with a peer controller pattern. The concurrency controller pattern includes Java classes to facilitate the behavior of a concurrency controller to be written in a guarded command fashion. The peer controller pattern includes peer controllers, helper Java classes to aid the implementation of asynchronous messaging for web services, and peer interfaces. Both of these verifiable patterns include a finite state machine implementation in Java to be used for writing behavioral interfaces. We formalize our interface

model as an extended finite state machine. We give formalisms for defining the semantics of both proposed design patterns.

• We have developed an assume-guarantee style modular verification strategy. To realize the behavior verification for concurrency controllers, we have implemented a translator that outputs a specification in the language of the model checker Action Language Verifier [23] from the concurrency controller classes and their interfaces. To realize the behavior verification for peer controllers, we have implemented a translator that outputs a specification in the language of the model checker SPIN [58] from the peer interfaces in a composite web service.

The interface verification in both application domains are performed with Java PathFinder [99] on isolated peers and threads. The peers are isolated through peer interfaces. For thread isolation we have implemented a data dependency analysis, generic stubs for modeling GUI components and network communication, a driver generator for modeling graphical user events, and an automated driver and stub generator for modeling RMI operations. The interactions among concurrent threads are modeled with controller interfaces and shared data stubs.

• We have applied the presented DFV approach for concurrent programs to two real-life software systems both of which have remote procedure calls and multiple threads. We have implemented a Concurrent Editor with about 2800 lines of code using the concurrency controller pattern. We have reengineered a safety critical air traffic control software, which consists of 21,057 lines of code, using the concurrency controller pattern. The study on this software system, called TSAFE, has empirical results demonstrating the effectiveness of our modular verification strat-

egy.

• We have applied the presented DFV approach for asynchronously communicating web services to four different web service implementations. Among these composite web services, two of them consist of five interacting peers and two of them consist of two interacting peers. We have implemented these web services based on the peer controller pattern and successfully verified both their interaction properties and the conformance of participant peers to their interfaces.

• We introduce a hierarchical state machine (HSM) model for specifying peer interfaces in a compact manner. The HSM model reflects the natural hierarchy of the service behavior and contains less number of states and transitions for peers that have concurrent executions of communication operations. We integrate the HSM model to the peer controller pattern. We extend the synchronizability analysis to HSMs so that we can identify synchronizable peer interfaces efficiently without flattening the HSMs.

Organization

The rest of the dissertation is organized as follows. Chapter 2 introduces the design patterns we have developed to apply the DFV principles to concurrent programming and to asynchronously communicating web services. Chapter 3 presents the formal models used in this dissertation. It includes our interface model, formal semantics of the presented design patterns, simple and abstract models for concurrent and distributed programs, the basis for verification methodologies, thread isolation, and the general interface verification approach. Chapter 4 presents the application

of DFV principles to concurrent programming. It introduces the DFV framework based on the concurrency controller pattern with the goal of eliminating synchronization errors in concurrent programs. Chapter 5 presents the DFV approach for developing reliable asynchronously communicating web services based on the peer controller pattern with the goal of automated verification of properties of interaction among the participating peers. Finally, Chapter 6 summarizes and concludes the dissertation.

Chapter 2

Design for Verification Patterns

This chapter presents two design patterns we have developed for applying the design for verification principles discussed in Chapter 1 to two different domains. The first pattern is called the *concurrency controller pattern*. This pattern is developed for applying our DFV approach to concurrent programming in Java with the goal of eliminating synchronization errors from Java programs using model checking techniques [11, 13, 16]. With this pattern, synchronization policies for coordinating the interactions among multiple threads are specified using concurrency controller classes instead of error-prone Java synchronization primitives. The second pattern is called the *peer controller pattern*. This pattern is developed for applying our DFV approach to asynchronously communicating web services [14, 15]. The goal here is to automatically verify properties of interactions among multiple peers participating in a web service implemented in Java.

In this chapter, we first present the concurrency controller pattern that enables verifiable concurrent programming in Java. Then, we continue with the presentation of the peer controller pattern that enables verifiable web service development in Java.

2.1 Verifiable Concurrent Programming

Reliable concurrent programming is especially important for Java programmers since threads are an integral part of Java. Efficient thread programming in Java requires conditional waits and notifications implemented with multiple locks and multiple condition variables with associated `synchronized`, `wait`, `notify`, and `notifyAll` statements. Concurrent programming using these synchronization primitives is error-prone with common programming errors such as nested monitor lockouts, missed or forgotten notifications, slipped conditions, and so forth [68].

We have developed a design for verification (DFV) framework based on a design pattern to facilitate verifiable concurrent programming in Java. The goal is to eliminate synchronization errors without sacrificing other desirable qualities of the software such as efficiency and maintainability. This pattern is called *concurrency controller pattern*. Using the concurrency controller pattern, a developer can write concurrency controller classes defining a synchronization policy without using any of the error-prone synchronization primitives of Java.

To implement a synchronization policy based on the concurrency controller pattern, a software developer needs to write a set of controller *actions* where each action is specified as a set of guarded commands. This is the behavior of the concurrency controller. The developer should also write an *interface* for the concurrency controller. The interface defines the allowed execution order of the actions for each thread. Note that, such an interface cannot be written as a Java interface; hence, we declare the controller interfaces as Java classes. The controller interface is a Java

class implementing a finite state machine with transitions representing controller actions.

In this section, we present the concurrency controller pattern in detail with an example. We also present an automated optimization for concurrency controllers to eliminate unnecessary context switches. The pattern presented in this section is the basis of the DFV approach we have developed for concurrent programming. The verification technique based on the concurrency controller pattern, however, will not be discussed in this section. Later, in Chapter 4, we show that both safety and liveness properties of concurrency controllers can be automatically verified using a modular verification technique.

2.1.1 Motivating Example

To illustrate the DFV approach for concurrent programs, we use the following example. Consider a data buffer implementation:

```
public class DataBuffer{
 private Vector data;
 ...
 public void put(Object in){...}
 public Object take(){...}
 public Object peek(){...}
}
```

Suppose that we want to share an instance of the `DataBuffer` class among multiple threads. If we declare all the methods of the `DataBuffer` class as `synchronized` to enforce mutual exclusion, a thread that calls `peek` method will be blocked by another thread executing the same method. One can achieve more efficient synchronization by using a reader-writer lock which allows multiple

threads to peek at the data buffer at the same time without blocking each other. However, a reader-writer lock may not be sufficient. For example, it may be necessary to implement a conditional wait for threads which call the `take` method when the buffer is empty. We may also want to put a bound to the size of the `DataBuffer` to prevent overflows or to limit the memory usage. In that case, it may be necessary to implement a conditional wait for threads which call the `put` method when the buffer is full.

To implement this synchronization strategy based on the concurrency controller pattern, we define a separate concurrency controller class which implements a bounded buffer protected by a reader-writer lock. The synchronization strategy implemented by this controller allows multiple threads to `peek` at the contents of the buffer at the same time, but it only allows a thread to perform a `put` or `take` operation when there is no other thread accessing the buffer. Additionally, this controller ensures that a thread that wants to put an item into the buffer waits while the buffer is full. Similarly, a thread that wants to take an item from the buffer waits while the buffer is empty. We call the controller which implements this synchronization BoundedBuffer-ReaderWriter (BB-RW) controller. The methods of the BB-RW controller are:

```
public interface BBRWInterface{
  public void w_enter_produce();
  public void w_enter_consume();
  public boolean w_exit();
  public void r_enter();
  public boolean r_exit();
}
```

Here, we define the return types of the methods `w_exit` and `r_exit` as booleans because we would like to give the flexibility of not releasing the read and write locks but we would like to know whether the lock is released or not.

To use the BB-RW controller in the above scenario, every thread that accesses the shared buffer instance should first invoke the related controller methods. For example, when a thread wants to put an item into the buffer, it needs to execute `w_enter_produce`, `put`, and `w_exit` methods in this order. This sequence guarantees that while performing the `put` operation, no other thread accesses the buffer and the buffer is not full. Similarly, when a thread wants to take an item from the buffer, it needs to execute `w_enter_consume`, `take`, and `w_exit` methods in this order. In our framework, these call sequences are checked during interface verification. Figure 2.1 shows an excerpt from a thread's code that illustrates this usage. Here, the Java interface of the BB-RW controller is used in the thread implementation.

In the next section, we explain the concurrency controller pattern, which aids the development of concurrency controllers such as the BB-RW controller, in detail.

2.1.2 Concurrency Controller Pattern

While developing multi-threaded programs that have a set of concurrently accessed shared data and that require conditional waits and notifications, the following *design forces* arise:

- *The implementation should be verifiable.* There should be a scalable automated verification framework which ensures that the implementations of concurrency controllers are correct with respect to desired safety and liveness

```
class MyClass extends Thread{
 private DataBuffer buffer;
 private BBRWInterface controller;

 public MyClass(DataBuffer buf, BBRWInterface c){
   buffer=buf; controller=c; ...
 }
 public void run (){
   ...
   controller.w_enter_produce();
   buffer.put(new Object());
   controller.w_exit();
   ...
   controller.r_enter();
   buffer.peek();
   controller.r_exit();
   ...
   controller.w_enter_consume();
   buffer.take();
   controller.w_exit();
   ...
 }
}
```

Figure 2.1: A sample thread that uses the BBRWController

properties. In recent years, there has been considerable progress in automated verification techniques for concurrent systems based on model checking [23, 99, 58, 72]. It should be possible to leverage this technology for the verification of concurrency controllers.

- *The implementation should avoid common concurrent programming errors.* Usage of error-prone synchronization statements such as `synchronized`, `wait`, `notify`, and `notifyAll` should be avoided to prevent common programming errors such as nested monitor lockouts, missed or forgotten notifi-

cations, and slipped conditions [68].

- *The synchronization strategy should be pluggable.* Encapsulating the synchronization strategy with the implementation of the shared data classes downgrades the reusability of the code. It is time consuming and error-prone to modify the code to support new synchronization strategies. Moreover, if the synchronization is encapsulated in the shared data classes, then manipulating more than one shared object at the same time could be difficult, or even impossible.

- *Shared data classes should be maintainable.* Encapsulating the implementation of the shared data with the synchronization operations makes modification of the shared data classes difficult. Managing synchronization among multiple threads and updating the states of the shared objects are separate concerns and it should be possible to modify them separately.

- *There should be an efficient mechanism to prevent unnecessary context-switch among threads.* The specific notification pattern [24] avoids context-switch overhead through multiple condition variables, multiple locks and notifications. However, the correct usage of these multiple locks and dependency analysis for correct notification is not easy to implement.

The concurrency controller pattern resolves these design forces. In this pattern, synchronization policies are implemented using guarded commands preventing error-prone usage of synchronization statements. The concurrency controller pattern separates the synchronization operations from the operations that change the shared

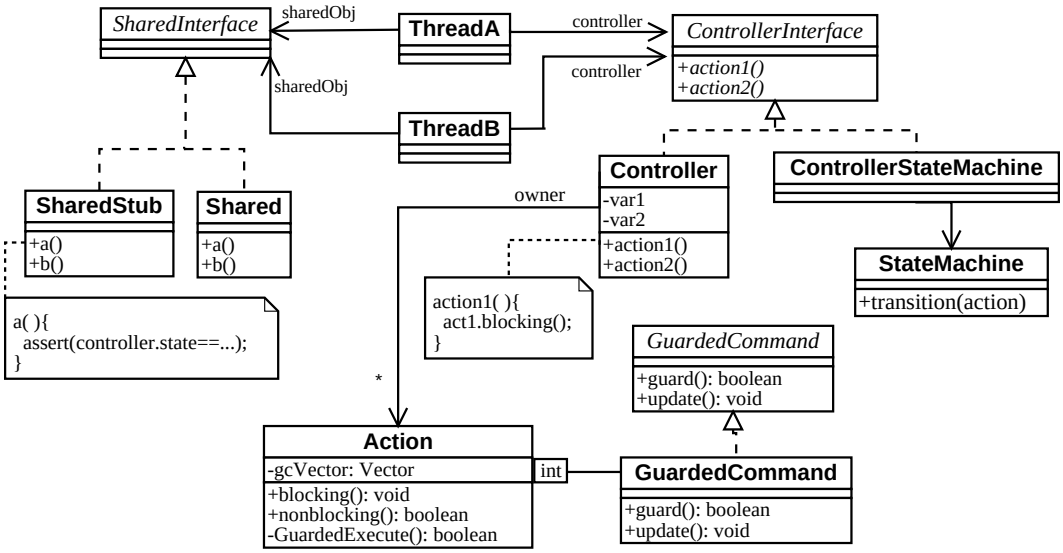


Figure 2.2: Concurrency controller pattern class diagram

object's internal state. This decoupling makes the synchronization policy pluggable and improves the maintainability of the code. We exploit this decoupling in our modular verification technique, which is presented in Chapter 4. The modularity improves the efficiency of the verification process and enables us to verify large systems by utilizing different verification techniques such as infinite state symbolic model checking and explicit state model checking with their associated strengths. The concurrency controller pattern also resolves the difficulty in efficient implementation of synchronization policies since we provide an automated optimization technique based on the specific notification pattern [24].

Figure 2.2 shows the class diagram for the concurrency controller pattern. The `ControllerInterface` is a Java interface that defines the names of actions available to user threads. The `Controller` class specifies the synchronization pol-

icy such as the BB-RW in Section 2.1.1. Multiple threads use an instance of the `Controller` class to coordinate their access to shared data. The `Controller-StateMachine` class is the controller interface that specifies the order of actions that can be executed by user threads. This class has an instance of the `StateMachine` class, which is a finite state machine implementation provided with the pattern and can be used as is. `SharedInterface` is the Java interface for the shared data such as the Java interface of the `DataBuffer`. The actual implementation of the shared data is the `Shared` class. The `SharedStub` class specifies the constraints on accessing the shared data based on the interface states of the concurrency controller.

Implementing The Concurrency Controller Behavior

A concurrency controller behavior is implemented with the `Controller` class in Figure 2.2. The variables of the `Controller` class store only the state information required for concurrency control. Each action of the `Controller` class is associated with an instance of the `Action` class and consists of a set of guarded commands. The code for the `Action` class is the same for each controller implementation. To write a concurrency controller class based on the pattern in Figure 2.2, a developer only needs to write the constructor for the `Controller` class, in which a set of guarded commands is defined for each action. Each method in the controller just calls the `blocking` or `nonblocking` method of the corresponding action. A blocking action causes the calling thread to wait until one of the guarding conditions becomes true whereas a nonblocking action does not cause the calling thread to wait. A nonblocking action returns true if a guarded command is executed

```
class BBRWController implements BBRWInterface{
 int nR; boolean busy; int count; int size;
 final Action act_r_enter, act_r_exit;
 final Action act_w_enter_produce, act_w_enter_consume, act_w_exit;
 BBRWController(int sz) {
  nR=0; count=0; busy=false; size=sz;
  Vector gcs = new Vector();
  gcs.add(new GuardedCommand() {
   public boolean guard(){ return (!busy );}
   public void update(){ nR = nR+1; } });
  act_r_enter = new Action(this,gcs);

  gcs = new Vector();
  gcs.add(new GuardedCommand() {
   public boolean guard(){ return true;}
   public void update(){ nR = nR-1; } });
  act_r_exit= new Action(this,gcs);

  gcs = new Vector();
  gcs.add(new GuardedCommand() {
   public boolean guard(){ return true;}
   public void update(){ busy = false; } });
  act_w_exit= new Action(this,gcs);

  gcs = new Vector();
  gcs.add(new GuardedCommand() {
   public boolean guard(){
     return (nR == 0 && !busy && count<size);}
   public void update(){ busy = true; count=count+1; } });
  act_w_enter_produce = new Action(this,gcs);

  gcs = new Vector();
  gcs.add(new GuardedCommand() {
   public boolean guard(){ return (nR == 0 && !busy && count>0 );}
   public void update(){ busy = true; count=count-1;} });
  act_w_enter_consume = new Action(this,gcs);
 }
 public void r_enter(){ act_r_enter.blocking();}
 public boolean r_exit(){return act_r_exit.nonblocking();}
 public void w_enter_produce(){ act_w_enter_produce.blocking();}
 public void w_enter_consume(){ act_w_enter_consume.blocking();}
 public boolean w_exit(){return act_w_exit.nonblocking();}
}
```

Figure 2.3: BB-RW implementation based on the concurrency controller pattern

and returns false if none of the guarding conditions were true.

Consider the BB-RW controller discussed in Section 2.1. A BB-RW controller can be implemented using four variables and five actions. The variables are `nR, busy, count`, and `size`. Here, `nR` denotes the number of readers in the critical section, `busy` denotes whether there is a writer in the critical section, `count` denotes the number of items in the buffer, and `size` denotes the size of the buffer. The actions are `r_enter`, `r_exit`, `w_enter_produce`, `w_enter_consume`, and `w_exit`. The controller class implementation for BB-RW controller is the `BBRWController` shown in Figure 2.3.

We specify the behavior of a concurrency controller in a guarded command style similar to that of CSP [57]. Since the Java language does not have a guarded command structure, we provide the `GuardedCommand` interface and the `Action` class. Each instance of the `Action` class has a vector of guarded commands that defines its behavior. The code for the `Action` class is given in Figure 2.4.

The `Action` class has three significant methods. The `GuardedExecute` method is used for executing one of the guarded commands of the action. If all the guards evaluate to false, then this method returns `false`. The execution of a blocking action is implemented by the `blocking` method. When a thread calls a blocking action, it has to execute a guarded command. Therefore, if the `GuardedExecute` method does not execute one of the guarded commands, then the thread waits in a loop, until it is notified by another thread. The execution of a nonblocking action is implemented by the `nonblocking` method. This method calls the `GuardedExecute`

```
public class Action{
 protected final Object owner;
 private final Vector gcV;
 public Action(Object c, Vector gcs){...}
 private boolean GuardedExecute(){
  boolean result=false;
   for(int i=0; i<gcV.size(); i++)
    try{
     if(((GuardedCommand)gcV.get(i)).guard()){
      ((GuardedCommand)gcV.get(i)).update();
      result=true; break; }
    }catch(Exception e){}
  return result;
 }
 public boolean nonblocking(){
  synchronized(owner) {
   boolean result=GuardedExecute();
   if (result) owner.notifyAll();
   return result; }
 }
 public void blocking(){
  synchronized(owner) {
   while(!GuardedExecute()) {
    try{owner.wait();}
    catch (Exception e){} }
   owner.notifyAll(); }
 }
}
```

Figure 2.4: Action class

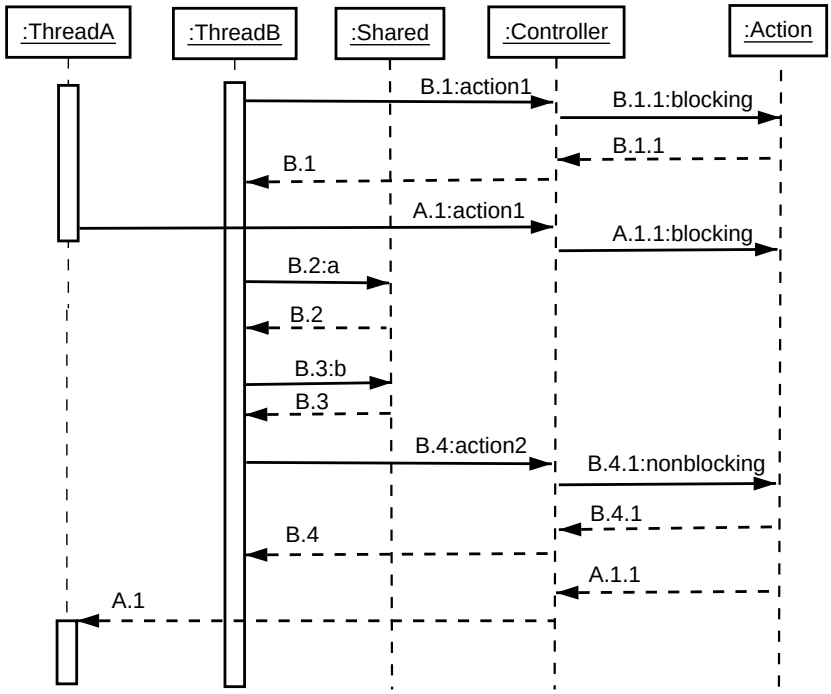


Figure 2.5: Controller sequence diagram

and notifies the other threads if a guarded command is executed. Note that, a nonblocking action does not cause the calling thread to wait. A nonblocking action returns `true` if a guarded command is successfully executed and returns `false` if the guards of all its guarded commands are false.

In a typical scenario, several threads would use an instance of a concurrency controller to coordinate their access to some shared data. Figure 2.5 shows a sequence diagram demonstrating the use of the concurrency controller pattern. In this scenario, thread `B` calls the controller action `action1`, which is a blocking action. After thread `B` executes the blocking action successfully, thread `A` calls the same

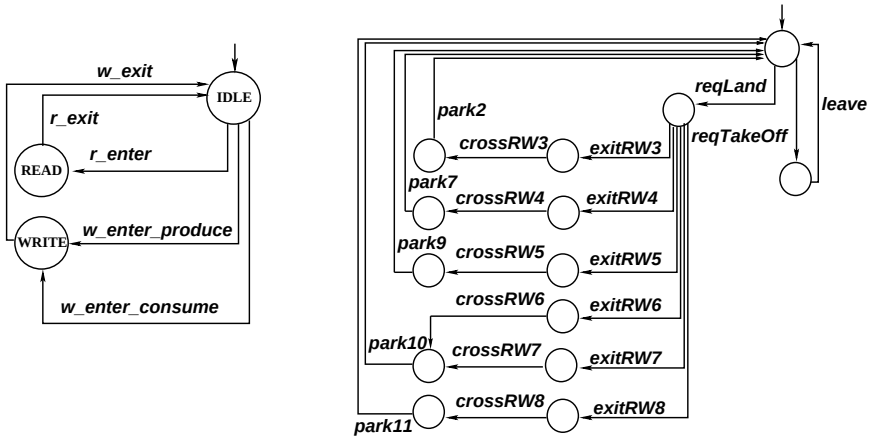


Figure 2.6: Concurrency controller interfaces

action, however, thread `A` is blocked by the controller. While thread `A` is blocked, thread `B` successfully executes a couple of operations on the shared object. After it finishes its operations on the shared object, thread `B` calls `action2` (a nonblocking action of the controller). The last controller action executed by thread `B` enables the action that is blocking thread `A`, and thread `A` successfully completes executing `action1`. This sequence of events, for example, corresponds to two threads interacting with each other using a mutex lock where `action1` corresponds to *acquire* action and `action2` corresponds to *release* action.

Implementing The Concurrency Controller Interface

The interface of a concurrency controller defines the acceptable call sequences for the threads that use the controller. Note that, controller interfaces have states and cannot be specified as Java interfaces. In the concurrency controller pattern, we use

Java classes to represent controller interfaces. The `ControllerStateMachine` class in Figure 2.2 defines the controller interface. This class encodes the state machine defining the allowed action call sequences. To encode the state machine we provide a `StateMachine` class, which is a finite state machine implementation and can be used as is. The `ControllerStateMachine` has the same set of methods as the concurrency controller itself. When a method of the controller interface is called, the transition method of the `StateMachine` with the corresponding action name is invoked. This `transition(action)` method first executes an assertion which checks that the current state is a state where the `action` can be executed, and then sets the current state to the target state of that transition.

The interface of the concurrency controller BB-RW is shown in Figure 2.6 (a). This interface has three states: `IDLE`, `READ`, and `WRITE`, with `IDLE` being the initial state. The interface state machine shows how the interface state changes when an action is executed. The BB-RW controller interface, for example, states that a thread using the BB-RW controller can execute (i.e. call) the `r_exit` action only after executing the `r_enter` action. This controller interface can be defined as `BBRWStateMachine` shown in Figure 2.7 by using a `StateMachine` instance. A method of this class contains only the invocation of the corresponding transition in the state machine instance, e.g., the `r_enter` method invokes the transition from `IDLE` to `READ`.

The controller interface is also used to specify when the methods of the shared data objects can be executed. For example, for the BB-RW controller protecting the `DataBuffer`, the `take` or `put` method of the `DataBuffer` can only be executed

```
public BBRWStateMachine implements BBRWInterface{
 StateMachine sm;
 final int IDLE=0; final int READ=1; final int WRITE=2;
 public BBRWStateMachine(int sz){
   sm=new StateMachine(3);
   ...
   sm.initial(IDLE);
   ...
   sm.addTransition("r_enter",IDLE,READ);
   sm.addTransition("r_exit",READ,IDLE);
   ...
 }
 public void r_enter(){ sm.transition("r_enter");}
 public boolean r_exit(){return sm.transition("r_exit");}
 ...
}
```

Figure 2.7: Parts of the controller interface implementation for BB-RW

in the `WRITE` state, the `peek` method can be executed in the `READ` and `WRITE` states, and no method of the `DataBuffer` can be executed in the `IDLE` state. In the concurrency controller pattern, these constraints are specified as assertions in a data stub class.

The interfaces of concurrency controllers can be complex. For example, the state machine in Figure 2.6(b) is the interface of a concurrency controller for an Airport Ground Traffic Control simulation program [105]. This controller consists of 13 integer variables and 20 actions. The controller actions are called for simulating the behavior of an airplane in an airport ground network model similar to that of the Seattle/Tacoma International Airport.

2.1.3 Optimizing Concurrency Controllers

Concurrency controllers written based on the design pattern given in Figure 2.2 may be inefficient because of the following reasons: 1) The pattern in Figure 2.2 does not use the specific notification [24], hence, after every state change in the concurrency controller all the waiting threads are awakened, increasing the context-switch overhead; 2) The inner classes used in the pattern in Figure 2.2 and the large number of method invocations may degrade the performance. To solve both of these problems, we automatically optimize the concurrency controllers using a source-to-source transformation. The optimized controller class 1) uses the specific notification pattern [24], 2) does not have any inner classes, and 3) minimizes the number of method invocations.

Implementing the specific notification pattern requires a notification dependency analysis which can be difficult and complicated to do manually. In our automated optimization process, we use the algorithm presented by Yavuz-Kahveci et al. [105] to compute these dependencies automatically. This algorithm is shown in Figure 2.8. We compute the notification dependencies and create a notification list for each action using this algorithm. In this algorithm, if the execution of the action a is able to make some guarding condition of action b true by updating some variable, then that action b is added to the notification list of a. To check this condition, we use pre- and post-condition computations provided in the Action Language Verifier. In the figure, the notations appear as in the original paper by Yavuz-Kahveci et al. [105]. In this notation, each action has one guarded command. (To handle actions with multiple guarded commands, we create temporary actions each of which has one

```
for each action a
    if d_s(a) ≠ true then
       mark a as guarded
       create condition variable cond_a
    else mark a as unguarded
    for each action b s.t. a ≠ b
        if POST(¬d_s(b), EXP(a)) ∩ d_s(b) ≠ ∅ then
           add cond_b to notifi cation list of a
```

Figure 2.8: Notification list computation algorithm by Yavuz-Kahveci et al. [105]

guarded command). Here, given an action a, $d_s(a)$ represents the guard condition of action a, EXP(a) represents the the conjunction of the guard condition and update expression, which is the definition of the action a.

To implement the specific notification pattern, we automatically generate one condition variable for each wait condition, i.e., for each blocking action. Condition variables are objects declared only for the purpose of synchronization. In the optimized concurrency controller class, when a thread is blocked while executing a blocking action, it waits on the condition variable of that action. Using a different condition variable for each blocked action improves the performance by awakening only the related threads.

Consider the actions of `BBRWController`. The notification list of `r_enter` is empty since the executions of these actions do not make any guarding condition true. The notification list of `r_exit` contains `w_enter_produce` and `w_enter_consume` actions. The `w_enter_produce` and `w_enter_consume` actions notify each other. Finally, the notification list of `w_exit` contains `r_enter`, `w_enter_produce` and `w_enter_consume` since its execution is able to make the guarding condition of

```
class BBRWController implements BBRWInterface{
 ...
 private boolean Guarded_w_enter_produce(){
  boolean result=false;
  synchronized(this) {
   if(nR==0 && !busy && count>size){
     busy=true; count=count+1; result=true;}
   else; }
  return result;
 }
 public void w_enter_produce(){
  synchronized(Condw_enter_produce){
   while (!Guarded_w_enter_produce()){
    try{ Condw_enter_produce.wait();
    } catch(InterruptedException e){}}}
  synchronized(Condw_enter_consume){
    Condw_enter_consume.notifyAll();
  }
 }
 public boolean w_exit(){ ...
  //notifies Condr_enter, Condw_enter_produce, Condw_enter_consume
 }
}
```

Figure 2.9: BBRWController class produced by optimization

these actions true. Figure 2.9 is an excerpt from the optimized version of BB-RW controller generated from the source given in Figure 2.3.

2.1.4 Related Work

Design patterns for multi-threaded systems have been studied extensively. For example, Schmidt et al. [88] present several interrelated patterns, including synchronization and concurrency patterns, for building concurrent and network systems. Some of these patterns, such as Active Object, Monitor Object and Strategized lock-

ing pattern, are closely related to our concurrency controller pattern. The object synchronizer [91] is another related pattern in the sense that it decouples the object functionality and synchronization management. Lea [68] also discusses several design patterns for concurrent object oriented programming and their usage in Java programs. All these patterns are built in order to help developers in writing reliable concurrent programs. Our goal, on the other hand, is to present a design pattern which improves the verifiability of concurrent programs by automated tools. In addition to presenting a verifiable design pattern for concurrency, we also present a modular verification technique that exploits the presented design pattern.

Lea [68] also provides a package of Java solutions for commonly used synchronization policies. Our concurrency controller implementations could be interpreted as a generalization of this framework. Our framework enables customized solutions for customized synchronization policies. A developer can write her own synchronization policy without much effort when she faces a new problem which requires a customized solution.

Deng et al. [35] propose a pattern system in their approach. The patterns in their system are idioms which are used for specifying a synchronization policy in a high-level language. These specifications are also used as abstractions when extracting the model of the program with the synthesized code to reduce the cost of automated verification. In our approach, we achieve the state space reduction during interface checking by replacing the controller implementations with the controller interfaces which serve as stubs. Although these approaches may seem similar, there are two important differences. The first difference is that, during behavior verification, we

can handle all ACTL properties including liveness properties, not just invariants. The other difference is that our approach is modular. During interface verification, we only check the correct usage of the concurrency controllers. Since the controller is guaranteed to satisfy the given synchronization properties, after behavior verification, interface verification does not have to search for synchronization errors and does not have to generate all possible interleavings of the concurrent threads.

The universe model presented by Behrends et al. [6] separates concurrency management from computation. The desired properties of a system are specified by universe invariants. Violations of these invariants are recognized at run-time. In our approach, verification is performed statically and programmers are not required to write specifications in another language.

To avoid the error-prone usage of low-level synchronization primitives, the recently released J2SE 5.0 [61] includes a concurrency utilities package [62, 65]. The package involves a `Lock` interface and a `ReadWriteLock` among other utilities. Similar to our framework, developers can create their own synchronization policies by implementing these interfaces. The verification approach enabled by the concurrency controller pattern can be adapted to automated verification of these custom implementations. With the concurrency utilities package, the lock acquisitions in the programs have to be explicit as well. Interface verification can be used to detect errors such as missing lock operations and unprotected data access.

In [78] a high-level inter-thread communication mechanism, called Message-Driven Thread API (MDT), is presented. The goal of this approach is to reduce the error-prone usage of synchronization primitives in Java. Unlike MDT, our approach

provides a framework for verification of the concurrent behavior of Java programs including correct usage of concurrency controllers with respect to their interfaces.

2.2 Verifiable Asynchronously Communicating Web Services

Web-based software applications that enable user interaction through web browsers have been extremely successful. Nowadays one can look for and buy almost anything online, from a book to a car, using such applications. A promising extension to this framework is the area of web services, i.e., web accessible software applications that interact with each other through the Internet. Web services have the potential to have a big impact on business-to-business applications similar to the impact interactive web software had on business-to-consumer applications.

Web services provide a framework for decoupling the interfaces of web accessible applications from their implementations, making it possible for the underlying applications to interoperate and integrate into larger, composite services. The following characteristics of web services are crucial for this purpose: 1) standardizing data transmission via XML [104], 2) loosely coupling interacting services through standardized interfaces, and 3) supporting asynchronous communication.

A fundamental problem in developing reliable web services is analyzing their interactions. The characteristics above present both opportunities and challenges in this direction. For example, decoupling of the interfaces and the implementations, which is necessary for interoperability, also provides opportunities for modular anal-

ysis. On the other hand, asynchronous communication, which is necessary to deal with pauses in availability of services and slow data transmission, makes analysis more difficult.

A composite web service consists of a collection of individual web services, called *peers*, working in a collaborative manner. Interaction among peers is established through *asynchronous messages*. In asynchronous communication, when a message is sent, it is inserted to a FIFO message queue, and the receiver consumes (i.e., receives) the message when it reaches the front of the queue. The interaction among the peers in a composite web service can be modeled as a *conversation*, the global sequence of messages that are exchanged among the peers [22, 51, 59]. A typical peer implementation includes the code for the operations specific to the application, the code for the asynchronous communication mechanism, and an interface specification describing the behavior of the peer.

We propose a behavioral design pattern called *peer controller pattern* for developing reliable web services. The peer controller pattern separates the operations related to the application logic from the communication details. The communication component is responsible for asynchronous messaging. The component implementing the application logic uses the communication component to interact with other peers. This decoupling improves the code maintainability and reusability and supports our modular verification strategy.

In the peer controller pattern, each peer has a behavioral interface description that captures the information needed by the other peers in order to interact with it. A *peer interface* is a Java class implementing a state machine which defines the order

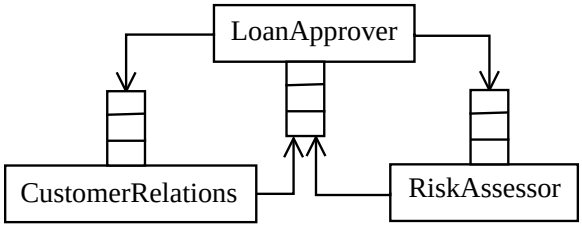


Figure 2.10: Loan Approval service

of send and receive operations that can be executed by that peer. The interface of a peer can be viewed as a contract between that peer and other peers which interact with it.

In this section, we discuss the peer controller pattern in detail. This pattern is the basis of the design for verification approach we developed for the verifiable web services. The peer controller pattern enables a modular, assume-guarantee style verification which is discussed in Chapter 5.

2.2.1 An Example Web Service

To illustrate the peer controller pattern we use the *Loan Approval* example described in the BPEL 1.1 specification [20]. In this example, a customer requests a loan for some amount. If the amount is small, the loan request is approved. For large amounts, a risk assessment service decides a risk level. The loan request is approved when the risk level is low and denied when the risk level is high.

The Loan Approval service is composed of three individual services (peers): CustomerRelations, LoanApprover and RiskAssessor (see Figure 2.10). Customers make loan requests using the CustomerRelations service. This service sends a *re-*

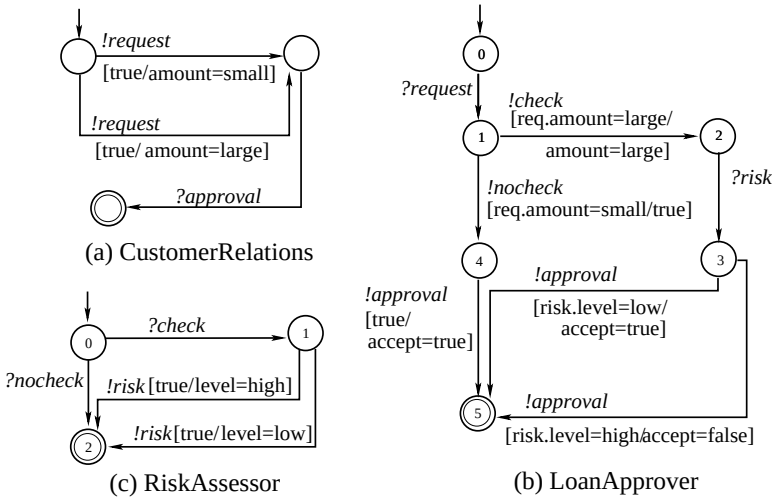


Figure 2.11: Peer interfaces

quest message to the LoanApprover service. The *request* message has a field called *amount*. If the request is for a small amount, the LoanApprover service sends an *approval* message, with the *accept* field set to true, to the CustomerRelations service. Otherwise, the LoanApprover service sends a *check* message to the RiskAssessor service. The RiskAssessor calculates a risk level and reports to the LoanApprover by a *risk* message with a *level* field. Then, the LoanApprover service sends an *approval* message to the CustomerRelations service with the *accept* field set to true or false depending on the content of the *risk* message received from the RiskAssessor service.

In this system, the communication among the peers is through asynchronous messaging. The Loan Approval service can process more than one customer application at a time. Each loan request generates a new session. The control logic

described above is the same for each session.

2.2.2 Peer Interfaces as Contracts

To reason about a composite web service, we need behavioral contracts describing the behaviors of the individual services, i.e., peers. We use finite state machines to specify behaviors of the peers and we call these state machines peer interfaces. Let us consider the Loan Approval example. Since this service is a composition of three services, one can specify the peer interfaces with three finite state machines, as shown in Figure 2.11.

The state machines in Figure 2.11 (a), (b), and (c) specify the behavioral interfaces of the CustomerRelations, LoanApprover and RiskAssessor services respectively. These behaviors are specified for one session. Here, *!message* denotes sending a message, *?message* denotes receiving of a message. There are 5 message types: *request* with an *amount* field, *approval* with an *accept*, *check* with an *amount*, *nocheck*, and *risk* with a *level* field. As seen in Figure 2.11, send transitions are labeled with conditions on the message contents. Consider the transition labeled with `!approval [risk.level=high/accept=false]` in Figure 2.11 (b). This transition is taken only if the `level` field of the last risk message is `high`. When this guarding condition holds, the LoanApprover peer sends an approval message with the `accept` field set to false.

2.2.3 Conversations

Using the peer interfaces, global behavior of a composite web service can be modeled as a set of state machines communicating with asynchronous messages, similar to the communicating finite state machine (CFSM) model. In [22, 46, 47] interactions among peers in such a system is specified as a conversation, i.e., the sequence of messages exchanged among peers, recorded in the order they are sent. A conversation is said to be complete if at the end of the session each peer ends up in a final state and each message queue is empty. (For simplicity, all conversations are assumed to be complete for the rest of the chapter). The notion of a conversation captures the global behavior of a composite web service where each peer executes correctly according to its interface specification, and every message ever sent is eventually consumed. (We assume that no messages are lost during transmission, which is a reasonable assumption based on the messaging frameworks provided by the industry [64, 77, 63]). For example, the following is a conversation that can be generated by the Loan Approval example in Figure 2: *request(amount=large)*, *check(amount=large)*, *risk(level=high)*, *approval(accept=false)*.

The conversation model gives us a convenient framework for reasoning about and analyzing interactions of web services. Given this framework, a natural problem is verifying properties related to conversations. As discussed in [46], temporal logic LTL can be extended to specify properties of conversations. A composite web service satisfies an LTL property if all the conversations generated by the service satisfy the property. We discuss how to perform behavior verification on conversations in Chapter 5 in detail.

2.2.4 Peer Controller Pattern

We propose the peer controller pattern for applying our DFV approach to web services to facilitate verifiable composite web service development. This pattern resolves the following design forces that arise in the development of reliable composite web services:

- *Contracts of peers should be explicit.* To achieve interoperability, the interface of a peer should be specified explicitly and should serve as a behavioral contract, specifying everything other peers need to know about a peer to interact with it. The interface of a peer should not be affected by the changes in the peer implementation that are not relevant to this contract.

- *The application logic of a peer should be implemented independent from the communication logic handling the asynchronous communication.* This separation is necessary for standardization of the communication and maintainability of the code.

- *The implementation should be amenable to automated verification.* Due to their distributed nature and asynchronous communication, web services are prone to errors. There should be a scalable automated verification framework to ensure their correctness.

The peer controller pattern resolves the above design forces. In the peer controller pattern, the application logic of a peer and the communication component are separated. This separation enables the developer to focus on the application logic

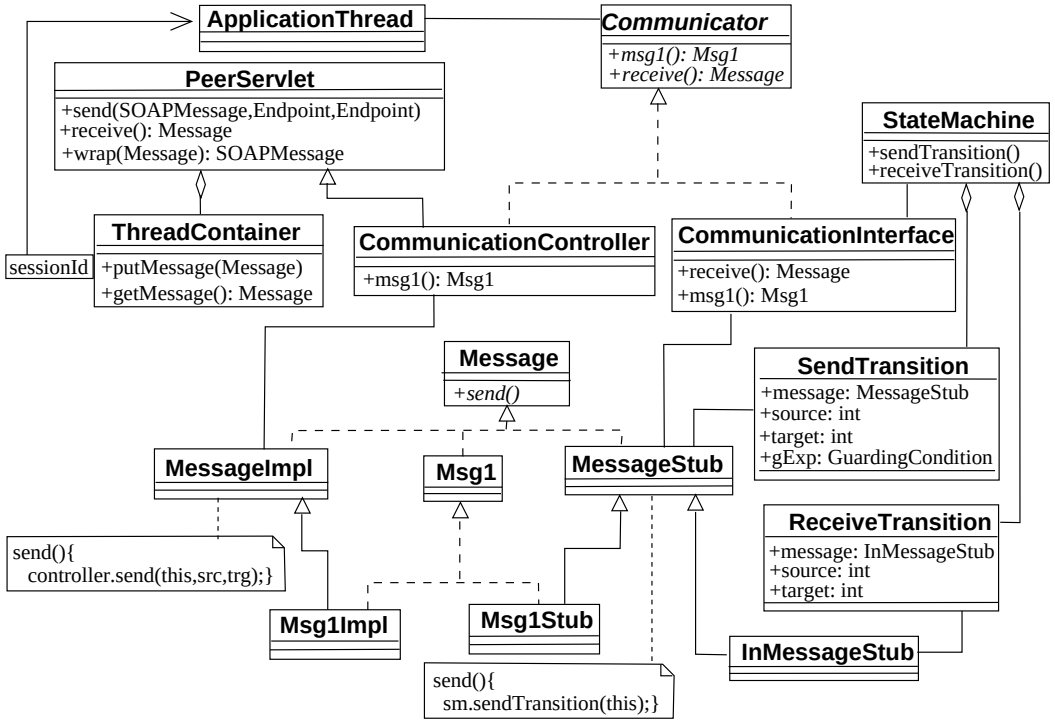


Figure 2.12: Class diagram for the peer controller pattern

without worrying about the details associated with the implementation of the asynchronous communication. This pattern also requires the developer to define the peer interface, which is the behavioral contract of the peer, explicitly. The peer interface is specified within the communication component. This explicit definition of the behavioral contract is crucial both for interoperability and modular verification.

The class diagram of the peer controller pattern is shown in Figure 2.12. The proposed pattern is session based. The application logic of a peer is the same for each session. This logic is implemented in the `ApplicationThread`. The application thread communicates asynchronously with other peers through the `Communicator`.

The `Communicator` is a Java interface that provides standardized access to the actual asynchronous communication implementation and its peer interface.

The actual communication is performed via the `CommunicationController` and customized message implementation classes (e.g. `Msg1Impl`). The peer interface, i.e., the behavioral contract for a peer, is written as a state machine via the `CommunicationInterface`. This class uses message stubs and an implementation of nondeterministic state machine (`StateMachine`).

Note that in the peer controller pattern, the communication component is more than a Business Delegator [30]. This component contains the behavioral contract of a peer, and plays a crucial role in verification.

Communication Controller

The `CommunicationController` class is a servlet that performs the actual communication. Since it is tedious to write such a class, we provide a servlet implementation (`PeerServlet`) that uses JAXM [63] in asynchronous mode. This helping servlet deals with opening an asynchronous connection, creating SOAP messages, and sending/receiving a SOAP message through JAXM provider. The `CommunicationController` class extends the `PeerServlet` and implements the `Communicator` interface. An interface method in this class returns a new actual message instance. Figure 2.13 shows a `CommunicationController` implementation for the LoanApprover peer.

The helping servlet is associated with a `ThreadContainer` that contains application thread references indexed by the session identifier. Whenever a message

```
public class ApproverServlet extends PeerServlet
  implements LAServlet{
 public void init(ServletConfig servletconfig)
       throws ServletException{
   super.init(servletconfig);
   urn="urn:LoanApprover";
 }
 public ApprovalMessage request(int sessionId){
   return new ApprovalMessageImpl(this,sessionId); }
 public CheckMessage check(int sessionId){
   return new CheckMessageImpl(this,sessionId); }
 public NoCheckMessage nocheck(int sessionId){
   return new NoCheckMessageImpl(this,sessionId); }
}
```

Figure 2.13: The ApproverServlet class for the LoanApprover peer

with a session identifier is received from the JAXM provider, it is delegated to the thread indexed with that session number. We use buffers for this message delegation. If there is no thread for the specified session, this container class creates a new application thread instance and starts that thread.

A sequence diagram is given in Figure 2.14 to explain the role of the `ThreadContainer` and a message reception. In this scenario, the JAXM provider receives a message from another peer. It delivers the message to the `CommunicationController` by calling the method `onMessage`. (The `onMessage` method is implemented within the `PeerServlet`. Recall that, this class is extended by the `CommunicationController`. In the figure, we show `PeerServlet` object instead of the controller since the JAXM provider interacts with it for the message reception.) The `onMessage` method invokes `putMessage` method of the `ThreadContainer`. The `ThreadContainer` looks for an application thread indexed with

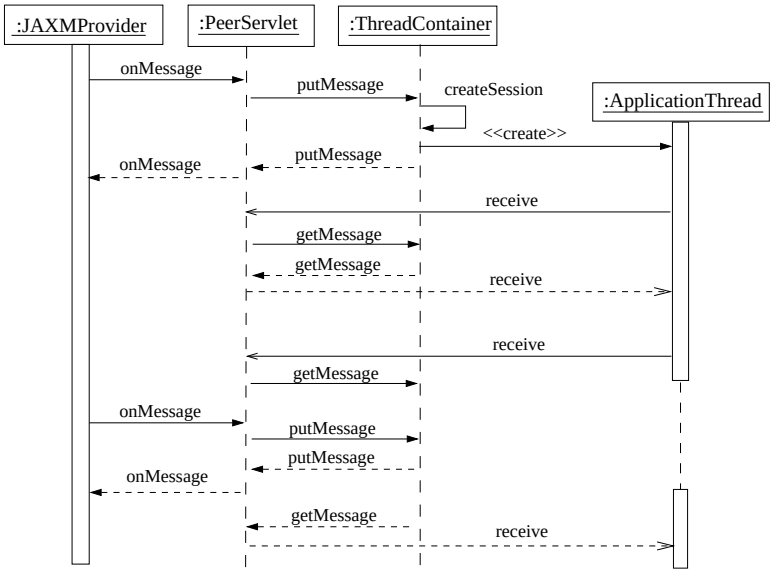


Figure 2.14: Sequence diagram for two message reception

the session identifier of the received message. In this scenario, there is no thread for the specified session. Therefore, a new `ApplicationThread` with this session identifier is instantiated. The message is stored in the buffer associated with this thread and the message reception by the JAXM provider is completed. Later, when the newly created `ApplicationThread` calls the `CommunicationController`, shown as the `PeerServlet` here, to receive a message, the first message from the buffer associated for that session is returned to the application thread. In the scenario shown in Figure 2.14, the newly created application thread invokes the communication controller to receive another message before the second message for this session is delivered to this peer. I.e. application thread tries to get a message and the buffer for this session is empty. The application thread stalls and notified by the JAXM

provider when a new message for that session is delivered.

Messages

The peers interact with each other using customized messages. The peer controller pattern requires each message type to be implemented in Java. This implementation consists of one actual message type definition, one message stub, and one Java interface to provide uniform access to these classes. The abstract class for the customized message types is shown as `MessageImpl` in Figure 2.12. Its `send` operation uses the sending method of the `PeerServlet`. The abstract class for the message stubs is `MessageStub` class. Its `send` operation uses the `sendTransition` method of `StateMachine`, which is explained below. This abstract class has a subclass that serves as a stub for incoming messages. Finally, the Java interface which unifies the actual message types and their stubs is called `Message` in Figure 2.12.

As an example, consider the *approval* message used by the LoanApprover peer. For this message type, we need to implement one Java interface, one message class that is used for communication, and one stub class that is used during verification. In Figure 2.12, these implementations correspond to `Msg1`, `Msg1Impl` and `Msg1Stub`, respectively.

In our framework, the attributes of message classes are categorized as control attributes and data attributes. The attributes that influence the behavior of the interface are called *control attributes*. A control attribute can be of enumerated or boolean type. Since at the time this framework build Java does not support enumerated types, we provide an `Enumerated` class. We use this separation between

control and data attributes in reducing the number of states of the message stubs. The stub for a message class only stores the values of the control attributes.

Communication Interface

The `CommunicationInterface` class is a special class that specifies the *peer interface*. A peer interface specifies the behavioral contract of the peer and it is also used during the verification process. The `CommunicationInterface` class contains two representative variables for each message type. One holds the last value and the other holds the current value of the control attributes of a message. These variables are the interface variables.

The `CommunicationInterface` encodes the state machine defining the contract by using the provided `StateMachine` class. In the constructor, the developer defines the transitions of the state machine. There are two kinds of transitions: send and receive transitions. A send transition is defined as a `SendTransition` instance. This instance stores the message, the source and the target states, and the guarding condition for that transition. A guarding condition consists of two conditions: the condition guardV specifies when the transition is executable and the condition guardP specifies the contents of the message to be sent with that transition. A guarding condition is defined as an anonymous inner class implementing the `GuardingCondition` Java interface.

The syntax of these conditions is defined as follows:

$guardV \rightarrow term \mid guardV$ && $guardV \mid guardV$ || $guardV \mid$!$guardV$

$guardP \rightarrow term \mid guardP$ && $guardP$

$term \rightarrow varname$ == $value$

In the condition guardV the *varname* is in the form of `last_msg.fieldname()` where `last_msg` is the name of the variable that holds the last value of the message `msg`. In this condition, the equalities are defined on the control fields of last messages. In the condition guardP, the equalities are defined on the message to be sent, i.e., *varname* is in the form of `msg.fieldname()`.

Consider the transition whose source state is 3, target state is 5, and labeled with `!`*`approval`*`[risk.level=high/accept=false]` in Figure 2.11(b). This send transition is implemented as

```
GuardingCondition gc=new GuardingCondition(){
  public boolean guardV(){
    return last_risk.level()==Low;}
  public boolean guardP(){
    return approval.accept()==true;}};
SendTransition outTrans= new SendTransition(approval,3,5,gc);
```

A receive transition is defined with a `ReceiveTransition` instance. This instance holds the message and the source and the target states. There is no guarding condition for receive transitions. In the LoanApprover peer, for example, the receive transition $(0,?$*request*$,1)$ in Figure 2.11(b) is specified with the statement

```
ReceiveTransition rcv= new ReceiveTransition(request,0,1);
```

where the variable `request` has the current value of the request message.

The `StateMachine` class is a nondeterministic finite state machine implementation. This class has two important methods: `sendTransition` and `receiveTransition`. The method `sendTransition(message)` computes the set of next interface states from the current interface states, asserts that this set is not empty. Then, it updates the current interface state, and saves the message instance to the corresponding interface variable. Note that only the values of control attributes are saved for a message instance. This set of next interface states is the target of the `SendTransition`s whose 1) label is of message type, 2) guarding conditions are satisfied, and 3) source state is in the current state. The `receiveTransition()` method computes the set of possible `ReceiveTransition`s available in the current interface state, and asserts that this set is not empty. If the set is not empty, it chooses one of these transitions nondeterministically. Here the nondeterminism is used to choose one type of message to receive. For example, consider a state from which two transitions with ?*msg1* and ?*msg2* are originated. With this nondeterminism, either ?*msg1* or ?*msg2* is chosen. Then, the method sets the current interface state, updates the interface variables, and returns the chosen incoming message stub instance. The incoming message stubs (`InMessageStub`) are generated in the preprocessing phase of the interface verification. These classes have an `instance` method that returns an instance with control field values chosen nondeterministically from the possible values that other peers can set. The nondeterminism is achieved by the `Verify.random` function which is a special function of the program checker Java PathFinder (JPF) [99]. This function forces JPF to search exhaustively every possible nondeterministic choice during interface verification.

2.2.5 Related Work

To achieve interoperability among web services a contractual agreement among participating peers is a necessity. The WSDL [103] standard is commonly used as a contract for specifying the operations, port types, and message types of an individual web service. This kind of information, however, is not sufficient for developing composite services. WSDL is a connectivity contract which does not model the behavior [76, 92]. A number of standards have been proposed for describing the behavior of a web service, such as BPEL [20], WSCI [102], and OWL-S (formerly DAML-S) [84]. Fu et al. [22] use state machines for this purpose and have shown that other behavioral descriptions (such as BPEL) can be translated to state machines [47]. Since state machines are powerful enough to specify the behaviors of web services and since they are suitable for automated reasoning, in our framework, the behavioral contracts among peers are specified as state machines.

State machines are used as behavioral contracts also by Gerede et al. [48] and Berardi et al. [9, 10]. Unlike our work, their goal is automatic web service composition. Also, Berardi et al. focus on sequences of activities performed rather than message sequencing in the composition. Benatallah et al. [8, 70] use statecharts to describe service behavior, specifically to declare a service composition. They present a framework for implementing web services without addressing verification of service interactions.

Currently, there are several design patterns to help developers in implementing web services [30, 1, 97, 7]. Fauvet et al. [42] introduce patterns for implementation and synthesis of composite services. Unlike these patterns, our goal is to present

a pattern which improves the verifiability of composite web services by automated tools. We also use the peer controller pattern to realize DFV for asynchronously communicating web services.

Chapter 3

Formal Models

In this Chapter we present the formal models used in this dissertation. We first define a general interface model. Then we introduce the formal semantics of the design patterns presented in Chapter 2: concurrency controller pattern and peer controller pattern. We continue with a simple and abstract model for concurrent programs, distributed programs, and asynchronously communicating distributed programs. After the presentation of these models, we define the formal basis for interface and behavior verification.

This chapter also addresses the thread isolation which is a form of environment generation problem [85, 96, 95]. We define how to isolate threads for interface verification and show that the interface verification can be performed separately on each thread.

We present a general interface verification approach and explain how to perform the interface verification directly on the implementation with a model checker called Java PathFinder based on the presented formalisms. Finally, we present a formalization of interface composition.

3.1 Interface Model

In this section we define our interface model. An interface of a controller specifies the allowed sequencing of controller actions and the assumptions of each action. An interface I is a finite state machine extended with interface variables. We denote an interface as a tuple $I = (Q, q_0, V, \Sigma, \delta, F)$ where Q is the set of finite states, $q_0 \in Q$ is the initial state, V is a finite set of typed interface variables each of which has a finite domain, Σ is the input alphabet, δ is the transition relation, and $F \subseteq Q$ is the set of final states. A configuration of I is defined as $c \in Q \times \prod_{v \in V} \text{DOM}(v)$. We denote the value of interface variables in a configuration c as $c(V)$ and the interface state as $c(Q)$. Given an interface I, there are a finite number of configurations since the sets Q and V are finite, and $\text{DOM}(v)$ for each $v \in V$ is finite.

Each element of Σ corresponds to a controller action. Each action $\sigma \in \Sigma$ has a finite set of control attributes denoted as $attr(\sigma)$. A control attribute of σ is an attribute that affects the behavior of the interface. While a general attribute of an action is unrestricted, the control attributes are restricted to have finite domains.

The transitions in δ are of the form (q, σ, g, u, q') where $q \in Q$ is the source state, $\sigma \in \Sigma$ is an action, g is the guard condition (defined below), $u : \prod_{v \in V} \text{DOM}(v) \rightarrow \prod_{v \in V} \text{DOM}(v)$ is the update function defined on interface variables, and $q' \in Q$ is the target state. Each transition has a guard condition g. A guard condition is a predicate of the form $g(attr(\sigma), c(V))$ where $\sigma \in \Sigma$ is an action and c is an interface configuration. The guard condition has two parts, $g = g_p \wedge g_v$. The condition g_p is a conjunction of predicates on $attr(\sigma)$. The condition g_v consists of predicates on

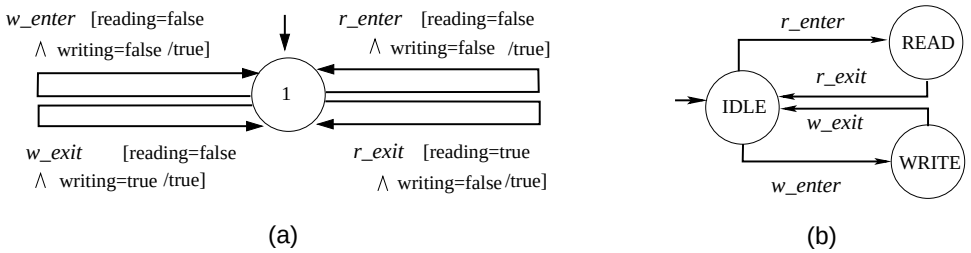


Figure 3.1: Two sample interface specifications

$c(V)$ combined with boolean operators. For brevity when the guard condition g is always TRUE, g is not shown in the transition. Similarly, when the update function u is identity function, u is not shown in the transition.

As an example, consider a reader-writer controller which allows multiple readers to access the data and only allows one writer to modify the data when there are no readers. One interface specification $I = (Q, q_0, V, \Sigma, \delta, F)$ for this controller is shown in Figure 3.1(a). The set of interface states is $Q = \{1\}$, the initial state is $q_0 = 1$, and the final state is $F = \{1\}$. The interface variables are $V = \{$`reading`,`writing`$\}$. The input alphabet is $\Sigma = \{$`r_enter, r_exit, w_enter, w_exit`$\}$. There are four transitions in this interface. One of these transitions is $(1,$`r_enter`$, g, u, 1)$ where the guarding condition $g = g_v \wedge g_p$ is $g_v \equiv$`writing=`FALSE $\wedge$ `reading=`FALSE and $g_p \equiv$ TRUE, and the update function u is $u($`reading,writing`$) =$ (TRUE, FALSE). The same order of actions for the reader-writer controller can be defined with another interface specification with no variables as shown in Figure 3.1(b). Note that both interfaces in Figure 3.1(a) and (b) specify the same action sequencing for the reader-writer controller.

An action sequence w is a legal sequence of $I = (Q, q_0, V, \Sigma, \delta, F)$ if the following conditions hold. First, each element σ of the sequence w should be an input symbol $\sigma \in \Sigma$. The second condition is defined by running the interface machine with the sequence w as follows. We start the interface machine at the initial configuration $c_0 = (q_0, v_{1_0}, v_{2_0}, \ldots, v_{n_0})$ where $n = |V|$ and v_{i_0} for $0 \leq i \leq n$ is the initial value the interface variable $v_i \in V$. We remove the first element σ of w from the sequence. If there is a transition $(c(Q), \sigma, g, u, q') \in \delta$ where c is the current configuration, and if $g(attr(\sigma), c(V))$ holds, the current configuration becomes $c' = (q', v'_1, v'_2, \ldots, v'_n)$ where $v'_1, v'_2, \ldots, v'_n$ store the values of interface variables in the configuration c' and $c'(V) = u(c(V))$. Otherwise, it is an error and the run stops. The execution continues with the removal of the new first element of w until w becomes empty or there is an error. When w is empty and the interface state in the current configuration is not a final state, there is an error. If there is no error and the first condition holds, then the sequence w is a legal sequence of I.

The transition relation δ is deterministic. Given a configuration c and an action σ there is only one next configuration. More precisely, given a configuration c where $c(Q) = q$ and an action σ, if there are any two transitions $(q, \sigma, g_1, u_1, q_1) \in \delta$ and $(q, \sigma, g_2, u_2, q_2) \in \delta$ where g_1 and g_2 evaluate to TRUE at c, then $c_1 = c_2$ where c_1 is the configuration after the first transition is taken and c_2 is the configuration after the second transition is taken. This determinism does not affect the expressiveness of the interface model since any nondeterministic finite state machine can be converted to an equivalent deterministic finite state machine.

3.2 Controller Semantics

In this section we first formalize the semantics of the concurrency controllers. We continue with a formalization of a composite web service specification based on the peer controller pattern. These semantic definitions are the formal model we use during behavior verification.

3.2.1 Concurrency Controller Semantics

A concurrency controller is a tuple $CC = (\Gamma, IC, A, I)$, where Γ is the set of controller variables, IC is the initial condition, A is the set of actions, and I is the controller interface. For example, for the concurrency controller BB-RW discussed in Section 2.1.1, $\Gamma = \{$ `nR`, `busy`, `count`, `size` $\}$ and $A = \{$ `r_enter`, `r_exit`, `w_enter_produce`, `w_enter_consume`, `w_exit` $\}$. The concurrency controller variables are private and can only be modified or accessed via the actions of CC. The initial condition denotes the initial values assigned to the concurrency controller variables in the constructor of the controller class. Formally, IC is a predicate on the concurrency controller variables in Γ, i.e., $IC : \prod_{\gamma \in \Gamma} \text{DOM}(\gamma) \rightarrow \{\text{TRUE}, \text{FALSE}\}$, where $\text{DOM}(\gamma)$ denotes the domain of the concurrency controller variable γ and $\prod_{\gamma \in \Gamma} \text{DOM}(\gamma)$ denotes the Cartesian product of the variable domains. For the BB-RW $IC \equiv \texttt{nR} = 0 \wedge \neg \texttt{busy} \wedge \texttt{count} = 0$.

The interface of a concurrency controller is a finite state machine $I = (Q, q_0, V, \Sigma, \delta, F)$ where Q is the set of states of the interface, $q_0 \in Q$ is the initial state of the interface, Σ is the input alphabet, and δ is the transition relation of the interface

(see Section 3.1). A controller interface specifies the execution order of controller actions and when the methods of the shared data can be executed. Therefore, Σ is the union of A and the methods of the shared data protected by the concurrency controller CC. The concurrency controller interface is a special case of the interface model presented in Section 3.1 in the sense that the set of final states has only the initial state ($F = \{q_0\}$). In the concurrency controller interfaces we investigated for the case studies in Chapter 4, there are no interface variables ($V = \emptyset$), no guard conditions, and no update functions. Therefore, in the rest of this chapter, for brevity, we will denote a transition in δ, because of the aforementioned features, as (q, act, q') instead of $(q, \mathit{act}, g, u, q')$ where $\mathit{act} \in A$, and $q, q' \in Q$. As an example, consider the interface of BB-RW given in Figure 2.6(a) in the previous chapter. At this interface $Q = \{$`IDLE`, `READ`, `WRITE`$\}$ and the initial state $q_0 =$`IDLE`. In this example, δ has five transitions and one of these transitions is (`IDLE`, `r_enter`, `READ`).

The semantics of a concurrency controller specification CC is a transition system $T(CC)(n) = (IT, ST, RT)$ where n is the parameter denoting the number of user threads, ST is the set of states, $IT \subseteq ST$ is the set of initial states, and $RT \subseteq ST \times ST$ is the transition relation. The transition system $T(CC)(n)$ represents all possible behaviors of a controller object when it is shared among n threads and assuming that each thread uses the controller object according to its interface. The initial states correspond to the states of the controller at the end of the execution of the constructor method. The transition relation RT represents the behavior of the controller object by recording its state at the end of the execution of each controller method that corresponds to an action.

The set of states is defined as the Cartesian product of the concurrency controller variable domains and the states of the user threads, $ST = \prod_{\gamma \in \Gamma} \text{DOM}(\gamma) \times \prod^{n} Q$. Note that, the state of a user thread is represented by an interface state and there is one interface state per user thread.

We introduce the following notation. Given a state $s \in ST$ and a controller variable $\gamma \in \Gamma$, $s(\gamma) \in \text{DOM}(\gamma)$ denotes the value of the variable γ in state s. Given a state $s \in ST$ and a set of variables $\Gamma' \subseteq \Gamma$, $s(\Gamma') \in \prod_{\gamma \in \Gamma'}$ denotes the projection of state s to the domains of the variables in Γ'. Finally, given a state s and a thread t, where $1 \leq t \leq n$, $s(Q)(t) \in Q$ denotes the state of thread t in s, and $s(Q - t) \in \prod^{n-1} Q$ denotes the projection of state s to the states of all the threads except thread t.

The initial states of the transition system $T(CC)(n)$ is defined as

$$IT = \{s \mid s \in ST \wedge IC(s) \wedge \forall 1 \leq t \leq n,\ s(Q)(t) = q_0\}$$

The set of actions, A, specifies the behavior of the concurrency controller. Each action *act* $\in A$, consists of a set of guarded commands *act*$.GC$. Each action has a blocking/nonblocking tag. For each guarded command $gc \in$ *act*$.GC$, guard $gc.g$ is a predicate on the variables Γ, $gc.g : \prod_{\gamma \in \Gamma} \text{DOM}(\gamma) \rightarrow \{\text{TRUE}, \text{FALSE}\}$. For each guarded command $gc \in$ *act*$.GC$, the update $gc.u$ is defined on controller variables $gc.u = \prod_{\gamma \in \Gamma} \text{DOM}(\gamma) \rightarrow \prod_{\gamma \in \Gamma} \text{DOM}(\gamma)$.

Consider a transition $(q, \textit{act}, q') \in \delta$ where q and q' are two interface states, and *act* is an action. We will define a transition relation $RT_{(q,act,q')} \subseteq ST \times ST$, which corresponds to executing the action *act* at interface state q. We define $RT^{u}_{(q,act,q')}$,

the transition relation for a transition (q, act, q') when one guarded command is executed, as

$$\begin{aligned} RT^{u}_{(q,act,q')} = \ & \{(s,s') \mid s,s' \in ST \wedge \ (\exists gc \in \mathit{act}.GC, gc.g(s(\Gamma)) \\ & \wedge s'(\Gamma) = gc.u(s(\Gamma))) \ \wedge \ (\exists 1 \leq t \leq n, s(Q)(t) = q \\ & \wedge s'(Q)(t) = q' \ \wedge \ s'(Q-t) = s(Q-t))\} \end{aligned}$$

If the action *act* is blocking, then the transition relation of (q, act, q') is defined as $RT_{(q,act,q')} = RT^{u}_{(q,act,q')}$. If the action a is nonblocking, then $RT_{(q,act,q')} = RT^{u}_{(q,act,q')} \cup RT^{nb}_{(q,act,q')}$ where $RT^{nb}_{(q,act,q')}$ denotes the case where none of the guards of *act* evaluate to TRUE

$$\begin{aligned} RT^{nb}_{(q,act,q')} = \ & \{(s,s') \mid s,s' \in ST \wedge \ s'(\Gamma) = s(\Gamma) \\ & \wedge (\forall gc \in \mathit{act}.GC, \ \neg gc.g(s(\Gamma))) \\ & \wedge (\exists 1 \leq t \leq n, s(Q)(t) = q \ \wedge \ s'(Q)(t) = q' \\ & \wedge s'(Q-t) = s(Q-t))\} \end{aligned}$$

The transition relation RT of the transition system $T(CC)(n)$ is defined as

$$RT = \bigcup\nolimits_{(q,act,q') \in \delta} RT_{(q,act,q')}$$

We define the execution paths of the transition system $T(CC)(n)$ based on RT as follows: An execution path $s_0, s_1, \ldots$ is a path such that $s_0 \in IT$ and $\forall i \geq 0, \ (s_i, s_{i+1}) \in RT$. Let AP denote the set of atomic properties, where a property $p \in AP$ is a predicate on the concurrency controller variables in Γ, $p : \prod_{\gamma \in \Gamma} \text{DOM}(\gamma) \rightarrow \{\text{TRUE}, \text{FALSE}\}$. We use ACTL to state properties of the transition system $T(CC)(n)$. A concurrency controller CC satisfies an ACTL formula f, if and only if, $\forall n \geq 0$, all the initial states of the transition system $T(CC)(n)$ satisfy the formula f.

3.2.2 Peer Controller Semantics

We focus on composite web services where the participant peers communicate with each other through asynchronous messaging. We denote the set of messages with M. We use $class(M)$ to denote the set of message classes and for each message $m \in M$ we use $class(m)$ to denote the message class of m. A composite web service specification is a tuple $W = (class(M), I_1, \ldots, I_k)$ where k is the number of peers in the composition, $class(M)$ is a finite set of message classes and I_i is the interface of the peer i for $1 \leq i \leq k$. For each message $m \in M$, *sender*$(m) \in \{I_1, \ldots, I_k\}$ denotes the peer that sends the message m, and *receiver*$(m) \in \{I_1, \ldots, I_k\}$ denotes the peer that receives the message m. We assume that there is one sender and one receiver for each message class.

Each peer interface $I_i = (Q_i, q_{0_i}, V_i, \Sigma, \delta_i, F_i)$ (see Section 3.1) is a finite state machine specifying the behavior of the peer i per session. We denote the configuration set of the interface I_i as $\mathcal{C}_i = Q_i \times \prod_{v \in V_i} \text{DOM}(v)$. The specialization in the interface model for peer interfaces is as follows. First, the transition relation δ is partitioned into send transitions δ_S and receive transitions δ_R. A receive transition represents the incoming calls to this peer and a send transition represents the outgoing calls from this peer. A receive transition is of the form $(q, ?\sigma, g, u, q') \in \delta_R$ where $q \in Q$ is the source state, $\sigma \in \Sigma$ is an action, g is the guard condition, u is the update function, and $q' \in Q$ is the target state. A send transition is of the form $(q, !\sigma, g, u, q') \in \delta_S$. The symbols $?$ and $!$ denote the receive and send transitions, respectively. Both send and receive transitions are triggered by an action.

The other specialization is that the input alphabet is $\Sigma = class(M)$, and each

interface variable $v \in V_i$ holds the content of a message. More precisely, $|V_i| = |class(M)|$ and for each control attribute f of each message class $mc \in class(M)$ there is an interface variable $v_{mc.f} \in V$ where $f \in attr(mc)$ that stores the last value of f. In other words, there is an interface variable that stores the last transmitted or received message instance of message class $class(m) \in class(M)$. Based on the peer controller pattern, the domain of each control attribute is finite which is the case for the interface variable definition in Section 3.1. Since the interface variables store the last exchanged message content, given a transition with a message $m \in M$, the update function u of this transition assigns the content of the message attributes to the corresponding interfaces variable v_m. The guard conditions are specified by the user as explained in Section 2.2.

As an example, consider the Loan Approver Service given in Section 2.2.1. In this example, there are three participant peers ($k = 3$). The peer interfaces are shown in Figure 2.11. Consider the interface of the LoanApprover peer. This interface has five interface variables since there are five message classes. The interface variable, for example, for the *risk* message class stores the control attributes of this message class. The *risk* message class has only *level* field as control attribute, i.e., $attr(\textit{risk}) = \{\textit{level}\}$ where $\text{DOM}(\textit{level}) = \{low, high\}$. In the LoanApprover peer interface δ_R has two and δ_S has five transitions. One of these transitions is $(3, !\textit{approval}, g, u, 5) \in \delta_S$ where the guard condition is $g = g_p \wedge g_v$ with $g_v \equiv v_{\texttt{risk.level}}$=`high` and $g_p \equiv$ `accept=false`, and the update function stores the values of the control attributes of the *approval* message into the corresponding interface variables.

The semantics of a composite web service specification is a transition system $T(W) = (IT, ST, RT)$ where ST is the set of states, $IT \subseteq ST$ is the set of initial states, and RT is the transition relation of the system. The set of states is defined as $ST = \mathcal{C}_1 \times \Theta_1 \times \mathcal{C}_2 \times \Theta_2 \times \ldots \mathcal{C}_k \times \Theta_k$ where k represents the number of peers in the composition, Θ_i is the set of configurations of the input queue of peer i, and $\mathcal{C}_i$ is the set of configurations of I_i.

We introduce the following notations. Given a state $s \in ST$ and a peer identifier i, $s(I_i)$ denotes the interface configuration of I_i in state s, and $s(\Theta_i)$ denotes the configuration of input queue of peer i in state s. We define two functions. The function *append* is used for manipulation of the queue configurations, where *append*(Θ_1, Θ_2) appends Θ_1 to the front of Θ_2. The function *first*(Θ) returns the first element in the Θ. $\langle\rangle$ denotes an empty queue and $\langle m \rangle$ where $m \in M$ denotes a queue containing a single message m.

The set of initial states of $T(W)$ is defined as

$$IT = \{s | s \in ST \ \wedge \ (\forall 1 \leq i \leq k, s(\Theta_i) = \langle\rangle \ \wedge \ s(I_i)(Q_i) = q_{0_i})\}$$

We define the following relation for the send transition $(q, !class(m), g, u, q')$.

$$\begin{aligned} RT_{(q,!class(m),g,u,q')} = \ & \{(s,s')|s,s' \in ST \wedge (\exists 1 \leq i \leq k,\ s(I_i)(Q_i) = q \\ & \wedge s'(I_i)(Q_i) = q' \ \wedge \ (q, !class(m), g, u, q') \in \delta_i \\ & \wedge g(attr(m), s(I_i)(V)) \ \wedge \ s'(I_i)(V) = u(s(I_i)(V)) \\ & \wedge (\forall 1 \leq j \leq k, j \neq i, s'(I_j) = s(I_j))) \\ & \wedge (\exists 1 \leq p \leq k,\ \textit{receiver}(m) = I_p \\ & \wedge s'(\Theta_p) = \textit{append}(s(\Theta_p), \langle m \rangle) \\ & \wedge (\forall 1 \leq l \leq k,\ l \neq p, s'(\Theta_l) = s(\Theta_l)))\} \end{aligned}$$

where $m \in M$.

We define the following relation for the receive transition $(q, ?class(m), g, u, q')$.

$$\begin{aligned} RT_{(q,?class(m),g,u,q')} = \ & \{(s,s')|s,s' \in ST \ \wedge \ (\exists 1 \leq i \leq k,\ s(I_i)(Q_i) = q \\ & \wedge s'(I_i)(Q_i) = q' \ \wedge \ (q, ?class(m), g, u, q') \in \delta_i \\ & \wedge g(attr(m), s(I_i)(V)) \ \wedge \ s'(I_i)(V) = u(s(I_i)(V)) \\ & \wedge (\forall 1 \leq j \leq k, j \neq i, s'(I_j) = s(I_j)) \\ & \wedge \textit{first}(s(\Theta_i)) = m \wedge \textit{append}(\langle m \rangle, s'(\Theta_i)) = s(\Theta_i) \\ & \wedge (\forall 1 \leq l \leq k, l \neq i, s'(\Theta_l) = s(\Theta_l)))\} \end{aligned}$$

where $m \in M$.

Finally, the transition relation RT for the $T(W)$ is defined as

$$\begin{aligned} RT = \ & \bigcup_{(q,!class(m),g,u,q') \in \delta_{S_i}, 1 \leq i \leq k} RT_{(q,!class(m),g,u,q')} \cup \\ & \bigcup_{(q,?class(m),g,u,q') \in \delta_{R_i}, 1 \leq i \leq k} RT_{(q,?class(m),g,u,q')} \end{aligned}$$

We define an execution sequence $exe = s_0, s_1, ...$ as a sequence of states where $(s_i, s_{i+1}) \in RT$. The conversation $conv(exe)$ generated by this sequence is defined

recursively as follows: The conversation $conv(s_0)$ is the empty sequence. The conversation $conv(s_0, s_1, ..., s_n, s_{n+1})$ is $conv(s_0, s_1, ..., s_n), m$ if there exists Θ_j such that $s_{n+1}(\Theta_j) = \textit{append}(s_n(\Theta_j), \langle m \rangle)$ and $m \in M$, or $conv(s_0, s_1, ..., s_n)$ otherwise. A conversation is a complete conversation if in the last state of the execution sequence each peer is in a final state and all the message queues are empty. We call the set of conversations generated by all the execution sequences of $T(W)$, the conversation set generated by $T(W)$.

3.3 Program Model

In this section, we first introduce a simple model for concurrent programs implemented based on the concurrency controller pattern. On this model we define two projection functions to formulate interface verification and behavior verification based on the concurrency controller pattern. We continue with a model for distributed programs that communicate via remote procedure calls and asynchronous messaging. Finally, we define a projection function on this model to define the behavior verification based on the peer controller pattern.

3.3.1 A Model for Concurrent Programs

In a single concurrent program P, there are three types of concurrent threads: 1) the main thread, 2) the threads that are created by other threads explicitly, and 3) the threads that are created implicitly by, for example, the Java Runtime Environment. In Java, an explicit thread is created with the invocation of the `start()` method of

a class that extends `java.lang.Thread` or implements `java.lang.Runnable`. For a single program, the only implicit thread is the event thread created by JVM (`java.awt.EventDispatchThread`) that dispatches the graphical user interface (GUI) events.

In the simple model presented in this section we have the following assumptions: 1) The shared variables are known and are implemented based on the concurrency controller pattern. Shared variables are only concurrency controllers and the shared data protected by these controllers. A variable is shared if more than one thread accesses that variable during a program execution. With an escape analysis [28, 19, 60] one can find this shared variable set. 2) The shared data is not primitive type. Otherwise, the concurrency controller pattern cannot be applied since the pattern requires an interface machine for the shared data. Also, the fields of the shared data are private and the only way to access or modify these fields is through method calls. 3) Suppose a thread t uses more than one controller (CC_1 and CC_2). We assume that t does not execute an action of CC_2 if it is not at the initial state (which is the only final state) of the interface of CC_1. However, we allow composing interfaces of concurrency controllers which then relaxes this restriction. Also, we assume that the controller variables of each concurrency controller are distinct.

Before formulating a program configuration and execution, we define the program stores in a single concurrent program. A shared store mapping is defined as $\rho \in \mathit{Sh} : \mathcal{V} \rightarrow \bigcup_{v \in \mathcal{V}} \textsc{dom}(v) \cup \perp$ where $\mathcal{V}$ is the set of program variables that are accessed by more than one thread. When a variable $v \in \mathcal{V}$ is mapped to $\perp$, it means that v is not visible to more than one thread. We assume that the mappings

are correct with respect to types. I.e., for any $\rho \in \mathit{Sh}$, $\rho(v) \in \text{DOM}(v) \cup \perp$ for all $v \in \mathcal{V}$. Based on the first assumption above, the elements of $\mathcal{V}$ to be used in shared store mappings are known and can be computed. Let v be an element of $\mathcal{V}$. A shared store mapping $\rho(v)$ returns the value of v if v is visible to more than one thread, and otherwise returns $\perp$. Intuitively, ρ represents the shared stores in P through which the concurrent threads communicate with each other. We define a local store mapping for a thread as follows. Given a thread t, the local store mapping of t is $\ell^t \in \mathit{Lcl}(t) : \mathcal{V} \rightarrow \bigcup_{v \in \mathcal{V}} \text{DOM}(v) \cup \perp$ where $\mathcal{V}$ is the set of program variables that are accessed by t. When a variable $v \in \mathcal{V}$ is mapped to $\perp$, it means that v is visible to more than one thread. We assume the local mappings are also correct with respect to types. Given $v \in \mathcal{V}$, a local store mapping $\ell^t(v)$ returns the value of v if v is visible only to t and returns $\perp$ otherwise. Intuitively, ℓ^t represents the local stores of t which are accesses only by t. Suppose $\ell^t(v) \neq \perp$ and $\rho(v) = \perp$ at a program configuration and v becomes visible to more than one thread at the next program configuration. Then the shared store mapping becomes $\rho'(v) \neq \perp$ and the local store mapping becomes $\ell'^t(v) = \perp$.

Configuration of a Single Program

Here we define a configuration of a single concurrent program implemented based on the concurrency controller pattern. A program configuration consists of a shared store mapping ($\rho \in \mathit{Sh}$) through which the threads interact with each other, a local store mapping for each thread t ($\ell^t \in \mathit{Lcl}(t)$) which is accessed only by t, a control state for each t ($\alpha^t \in \mathit{Ctl}(t)$) which is the program counter of t. Formally, the set of

program configuration is $\{c | c \in \mathit{Sh} \times \prod_{t \in T}(\mathit{Lcl}(t) \times \mathit{Ctl}(t))\}$ where T is the finite set of threads. The number of threads in a program changes during the program execution with explicit thread creations and thread terminations. We represent the set of threads at a configuration as $c(T)$. We assume that all of the implicit threads are created at the program start. When a single concurrent program starts, there are two threads: the main thread t_m and the event thread t_e. A program starts with an initial configuration $c_0 = (\rho_0, \ell_0^{t_m}, \alpha_0^{t_m}, \ell_0^{t_e}, \alpha_0^{t_e})$ where $\rho_0 \in \mathit{Sh}$ is the initial shared store mapping that returns $\perp$ for every variable, $\ell_0^{t_m} \in \mathit{Lcl}(t_m)$ is the initial local store mapping of the main thread t_m, $\alpha_0^{t_m} \in \mathit{Ctl}(t_m)$ is the initial control state of t_m, $\ell_0^{t_e} \in \mathit{Lcl}(t_e)$ is the initial local store mapping of the event thread t_e, $\alpha_0^{t_e} \in \mathit{Ctl}(t_e)$ is the initial control state of t_e. The program configuration changes with an input event or with the execution of one operation by one of the concurrent threads. Next, we define the input events and operations.

Operations and Input Events

Given a thread t, an operation that t can execute is represented as $op^t(a)$ where $a = a^0, a^1, \ldots, a^l$ is the argument sequence of op. (To simplify the discussion, we assume that these operations are performed by method calls, which is a reasonable assumption in object oriented programming.) The argument a^0 holds the return value of the method call and a^1 holds the exceptions thrown during the operation. There are five types of operations: local operations, shared operations, thread creation, thread termination, and environment interaction operations. We will use op to denote any of these kinds of operations.

Shared operations are the operations through which threads interact with each other. These operations are either a read from a shared variable or a write to a shared variable. Recall that a shared variable can be either a concurrency controller instance or a shared data protected by a controller. Based on the concurrency controller pattern, all of the shared operations are known beforehand. A shared operation performed by t with an argument sequence a is defined as $sop^t(a) : \mathit{Sh} \times \mathit{Ctl}(t) \rightarrow \mathit{Sh} \times \mathit{Ctl}(t)$.

The thread creation operation is a special type of interaction that is used for the creation of explicit threads. Let t be the thread that creates the explicit thread t' with the operation $tcop^t(a)$ where a is the sequence of arguments passed to t'. We define this type of operation as $tcop^t(a) : \mathit{Sh} \times \mathit{Lcl}(t) \times \mathit{Ctl}(t) \rightarrow \mathit{Sh} \times \mathit{Lcl}(t) \times \mathit{Ctl}(t) \times \mathit{Lcl}(t') \times \mathit{Ctl}(t')$. With a thread creation operation there are zero or more program variables that become visible to more than one thread. Let $tcop^t(a)(\rho_i, \ell_i^t, \alpha_i^t) = (\rho_{i+1}, \ell_{i+1}^t, \alpha_{i+1}^t, \ell_0^{t'}, \alpha_0^{t'})$ be a thread creation operation where $\rho_i, \rho_{i+1} \in \mathit{Sh}$, $\ell_i^t, \ell_{i+1}^t \in \mathit{Lcl}(t)$, $\alpha_i^t, \alpha_{i+1}^t \in \mathit{Ctl}(t)$, $\ell_0^{t'} \in \mathit{Lcl}(t')$ is the initial local store mapping for thread t', and $\alpha_0^t \in \mathit{Ctl}(t')$ is the initial control state of t'. Let also $v \in \mathcal{V}$ be a program variable where $\ell_i^t(v) \neq \perp$ and $\rho_i(v) = \perp$. If v is an argument passed to t', then $\ell_{i+1}^t(v) = \perp$ and $\rho_{i+1}(v) = \ell_i^t(v)$. Note that, the only influence of a thread creation operation on the shared variables is to change their visibility. Another effect of a thread creation operation is the change in the number of threads within the program configuration. Let $tcop^t(a)$ be a thread creation operation, c be the program configuration just before $tcop^t(a)$, and c' be the program configuration just after the execution of this operation. Then the set of threads at the next configuration is $c'(T) = c(T) \cup \{t'\}$.

Also, c' contains the local store mappings and the control state of all the threads existing in c and the new created thread ($Lcl(t')$ and $Ctl(t')$). The other type of operation that changes the number of threads in a program is the termination operations ($tdop^t$). If c and c' are the program configurations right before and after the execution of a termination operation $tdop^t$, respectively, then $c'(T) = c(T) \setminus \{t\}$.

Environment interaction operations are the operations that a thread performs to communicate with its environment and that are not shared operations. The environment interaction operation types are 1) GUI operations, 2) file read and write operations, 3) socket operations, and 4) command line argument read. For example, a GUI operation is an invocation of a method of a GUI object. However, if this GUI object is shared then any method call to this object is classified as a shared operation. The environment interaction operations do not affect the shared variables directly. A thread can reflect the effect to the shared variables with shared operations after an environment interaction. An environment interaction operation performed by t is defined as $eop^t(a) : Lcl(t) \times Ctl(t) \rightarrow Lcl(t) \times Ctl(t)$.

The other form of environment interaction is input events. The input events are triggered outside of the program. For example, a button click event is an input event that is triggered by a user and delivered to the program. We denote an input event as $e(ea)$ where ea is the attribute sequence of the event. The input events for a single program are the GUI events. An input event has zero or more attributes. These events are handled by the event thread t_e; therefore, a GUI event affects the configuration of t_e. Since the input events are triggered outside of the program, we assume that they do not affect the shared variables directly. These events change

the local store of t_e and t_e can later alter the shared variables by performing shared operations. The formal definition of an input event for single programs is $e(ea)$: $\mathit{Lcl}(t_e) \times \mathit{Ctl}(t_e) \rightarrow \mathit{Lcl}(t_e) \times \mathit{Ctl}(t_e)$.

Local operations are the operations that change only a thread's local store mapping and its control state, and that are not environment interaction operations. Given a thread t, a local operation performed by t is defined as $lop^t(a) : \mathit{Lcl}(t) \times \mathit{Ctl}(t) \rightarrow \mathit{Lcl}(t) \times \mathit{Ctl}(t)$.

Execution of a Single Program

We will define the transitions in a single program P with a relation $R_P : \mathcal{C} \times \mathcal{C}$ where $\mathcal{C}$ is the set of configuration in P. These transitions occur with an input event or with the execution of one operation by one of the concurrent threads. Let $(c, c') \in R_P$ be a transition of the program P with an operation op^t or event e. The configuration c is the program configuration just before op^t (or e), and the configuration c' is the program configuration just after the completion of op^t (or e). Below we define a number of transition relations for each operation types and input events to aid the definition of R. Then we give the definition of R based on these relations.

We introduce the following notations. Given a program configuration c, $c(\mathit{Sh}) \in \mathit{Sh}$ denotes the shared store mapping at the configuration c. Given a thread $t \in T$ and a program configuration c, $c(\mathit{Lcl}(t)) \in \mathit{Lcl}(t)$ denotes the local store mapping of t and $c(\mathit{Ctl}(t)) \in \mathit{Ctl}(t)$ denotes the control state of t at the configuration c.

We define R_{sop^t}, the transition relation for the execution of a shared operation by

a concurrent thread t with the argument sequence a, as

$$\begin{aligned} R_{sop^t(a)} = \quad & \{(c, c') | c, c' \in \mathcal{C} \ \wedge \ sop^t(a)(c(\mathit{Sh}), c(\mathit{Ctl}(t))) = (\rho, \alpha^t) \\ & \wedge c'(\mathit{Sh}) = \rho \ \wedge \ c'(\mathit{Ctl}(t)) = \alpha^t \ \wedge \ c'(\mathit{Lcl}(t)) = c(\mathit{Lcl}(t)) \\ & \wedge (\forall t' \in c(T), \ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \\ & \wedge c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

Recall that a shared operation is either a concurrency controller action execution or a shared data method invocation. As discussed in Section 3.2.1, a thread t can be blocked while executing a concurrency controller action. Let $sop^t(a)$ be such a shared operation. In this case, the transition with this shared operation is not taken until the thread t completes the action. Meanwhile, other threads of the program change the program configuration.

We define R_{lop^t}, the transition relation for the execution of a local operation by a concurrent thread t with the argument sequence a, as

$$\begin{aligned} R_{lop^t(a)} = \quad & \{(c, c' | c, c' \in \mathcal{C} \ \wedge \ lop^t(a)(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = (\ell^t, \alpha^t) \\ & \wedge c'(\mathit{Sh}) = c(\mathit{Sh}) \ \wedge \ c'(\mathit{Ctl}(t)) = \alpha^t \ \wedge \ c'(\mathit{Lcl}(t)) = \ell^t \\ & \wedge (\forall t' \in c(T), \ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \\ & \wedge c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

Recall that, local operations do not affect the shared variables. Therefore, modeling a local operation execution as an atomic execution does not influence the correctness of the model with respect to synchronization behavior.

We define R_{eop^t}, the transition relation for the execution of an environment in-

teraction operation by a concurrent thread t with the argument sequence a, as

$$\begin{aligned} R_{eop^t(a)} = \quad & \{(c, c' | c, c' \in \mathcal{C} \ \wedge \ eop^t(a)(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = (\ell^t, \alpha^t) \\ & \wedge c'(\mathit{Sh}) = c(\mathit{Sh}) \ \wedge \ c'(\mathit{Ctl}(t)) = \alpha^t \ \wedge \ c'(\mathit{Lcl}(t)) = \ell^t \\ & \wedge (\forall t' \in c(T), \ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \\ & \wedge c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

For thread creation operation we define the transition relation R_{tcop^t}, where t is the thread performing the operation $tcop$ and t'' is the thread created with that operation. This definition is as follows.

$$\begin{aligned} R_{tcop^t(a)} = \quad & \{(c, c' | c, c' \in \mathcal{C} \\ & \wedge tcop^t(a)(c(\mathit{Sh}), c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = (\rho, \ell^t, \alpha^t, \ell_0^{t''}, \alpha_0^{t''}) \\ & \wedge c'(\mathit{Sh}) = \rho \ \wedge \ c'(\mathit{Lcl}(t)) = \ell^t \ \wedge \ c'(\mathit{Ctl}(t)) = \alpha^t \\ & \wedge c'(\mathit{Lcl}(t'')) = \ell_0^{t''} \ \wedge \ c'(\mathit{Ctl}(t'')) = \alpha_0^{t''} \ \wedge \ c'(T) = c(T) \cup \{t''\} \\ & \wedge (\forall t' \in c(T), \ t' \neq t \ \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \\ & \wedge c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

We define R_{tdop^t}, the transition relation for termination operation as

$$\begin{aligned} R_{tdop^t(a)} = \quad & \{(c, c' | c, c' \in \mathcal{C} \ \wedge \ tdop^t(a)(c(\mathit{Sh}), c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = (\rho) \\ & \wedge c'(\mathit{Sh}) = \rho \ \wedge \ c'(T) = c(T) \setminus \{t\} \\ & \wedge (\forall t' \in c(T), \ t' \neq t \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \\ & \wedge c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t')))\} \end{aligned}$$

Based on these definitions, we define the transition relation for an operation op with argument sequence a which is performed by a thread t as

$$R_{op^t(a)} = R_{sop^t(a)} \cup R_{lop^t(a)} \cup R_{eop^t(a)} \cup R_{tcop^t(a)} \cup R_{tdop^t(a)}$$

We define a transition relation $R_{e(ea)}$ which corresponds to the execution of the input event e with an attribute sequence ea as follows.

$$\begin{aligned} R_{e(ea)} = \ & \{(c, c' | c, c' \in \mathcal{C} \ \wedge \ e(ea)(c(\mathit{Lcl}(t_e)), c(\mathit{Ctl}(t_e))) = (\ell^{t_e}, \alpha^{t_e}) \\ & \wedge c'(\mathit{Sh}) = c(\mathit{Sh}) \ \wedge \ c'(\mathit{Lcl}(t_e)) = \ell^{t_e} \ \wedge \ c'(\mathit{Ctl}(t_e)) = \alpha^{t_e} \\ & \wedge (\forall t \in c(T), \ t \neq t_e \Rightarrow c'(\mathit{Lcl}(t)) = c(\mathit{Lcl}(t)) \\ & \wedge c'(\mathit{Ctl}(t)) = c(\mathit{Ctl}(t)))\} \end{aligned}$$

Finally, the transition relation R^P is the union of all of the input event and operation execution transition relations for all of the input events in the program and operations performed by all of the threads.

$$R^P = \bigcup_{op^t(a) \in OP^t, t \in T} R_{op^t(a)} \cup \bigcup_{e(ea) \in E} R_{e(ea)}$$

where T is the set of threads, E is the set of events for all event attribute sequences, and OP^t is the set of operations that a thread t can perform with any argument sequences.

An execution of program P is defined as follows. P starts with the initial configuration $c_0 = (\rho_0, \ell_0^{t_m}, \alpha_0^{t_m}, \ell^{t_e}, \alpha_0^{t_e})$. The program configuration is updated with input events and operations according to the transition relation R_P defined above. We represent one step program execution with an input event e whose attribute sequence is ea as $c \xrightarrow{e(ea)} c'$ where $(c, c) \in R_{e(ea)}$. Similarly, we denote one step program execution with an operation op whose argument sequence is a and performed by thread t as $c \xrightarrow{op^t(a)} c'$ where $(c, c) \in R_{op^t(a)}$. A program execution $x_p = x_0, x_1, \ldots, x_i, x_{i+1}, \ldots$ is a sequence where $x_0 = c_0 \xrightarrow{op_0^{t_m}(a_0)} c_1$ and each element x_i in this sequence is of the form $x_i = c_i \xrightarrow{label} c_{i+1}$ with $(c_i, c_{i+1}) \in R^P$ and $label$ is an operation or input event.

3.3.2 Projections on Concurrent Programs

Having defined the execution of concurrent programs, we now define two projection functions so that we can form the basis for interface and behavior verifications.

First we introduce a product interface machine I^p for concurrency controller interfaces. Given k controller interface instances $I_1, I_2, \ldots, I_k$ of k concurrency controller instances where $I_i = (Q_i, q_{0_i}, \{\}, \Sigma_i, \delta_i, F_i)$ and $1 \leq i \leq k$, we define the product machine as $I^p = (Q^p, q_0^p, V^p, \Sigma^p, \delta^p, F^p)$. The set of states in I^p is $Q^p = Q_1 \times Q_2 \times \cdots \times Q_k$. The initial state of I^p is $q_0^p = (q_{0_1}, q_{0_2}, \ldots, q_{0_k})$. The set of interface variables of this product machine is $V^p = \bigcup_{1 \leq i \leq k} V_i$, the input alphabet is $\Sigma^p = \bigcup_{1 \leq i \leq k} \Sigma_i$, and the set of final states is $F^p = \{q_0^p\}$. The transition relation δ^p is defined as follows. Given a state $q \in Q^p$ and an integer $1 \leq i \leq k$, let $q[i] \in Q_i$ denote the ith element of q. The transition (q, σ, q') is in δ^p if and only if $q, q' \in Q^p$ and $\exists 1 \leq i \leq k,\ s.t.\ (q[i], \sigma, q'[i]) \in \delta_i\ \wedge\ (\forall 1 \leq j \leq k, j \neq i \Rightarrow q[j] = q'[j])$. Note that, this product machine encodes the last assumption stated at the beginning of Section 3.3.1.

Here we introduce our first projection function $\Pi_1 : X_p \times T \rightarrow SOp$ where X_p is the set of all program executions, T is the set of threads, and SOp is a set of shared operation sequences. Given a program execution $x_p = x_0, x_1, \ldots$ where $x_p \in X_p$ and a thread t, the function $\Pi_1(x_p)(t)$ removes from x_p the configurations, input events, and the operations other than the shared operations performed by t. The projection $\Pi_1(x_p)(t)$ is formally defined recursively as follows. Let c_j be the first program configuration from which t performs a shared operation. I.e., the shared operation at $c_j \xrightarrow{sop_0^t(a_0)} c_{j+1}$ is the first shared operation per-

formed by t. The projection $\Pi_1(x_0, \ldots, x_{j-1})$ is the empty sequence. The projection $\Pi_1(x_0, \ldots, x_j)$ is $sop_0^t(a_0)$. The projection $\Pi_1(x_0, \ldots, x_j, \ldots, x_{l+1})$ for $l > j$ is $\Pi_1(x_0, \ldots, x_j, \ldots, x_l), sop_z^t(a_z)$ if x_{l+1} is $c_{l+1} \xrightarrow{sop_z^t(a_z)} c_{l+2}$. Otherwise, the projection is $\Pi_1(x_0, \ldots, x_j, \ldots, x_{l+1}) = \Pi_1(x_0, \ldots, x_j, \ldots, x_l)$.

Recall that, based on the design for verification approach, the shared operations are known and explicit in the program. Moreover, these operations are either controller actions or shared data operations since only concurrency controllers and the shared data protected by these controllers are accessed by more than one thread based on the concurrency controller pattern. Therefore, the result of this projection function is a sequence of controller actions and accesses to the shared data protected by these controllers in the order they are performed by t.

Using the product machine definition I^p and the projection Π_1 we give the following definition to be used as the *correctness criteria* during the interface verification.

Definition 3.3.1 (Thread Interface Correctness) *Let $I_0, \ldots I_k$ be the interfaces of the concurrency controllers used by a thread t and I^p be the product machine of these interfaces. The thread t is* **interface correct** *if for all $x_p \in X_p$, $\Pi_1(x_p)(t)$ is a legal sequence of the product machine I^p.*

Before introducing the projection function to be used to form the basis for behavior verification of a concurrency controller, we define thread obedience to the interface of a controller. First we extend the definition of Π_1 with a concurrency controller attribute. Given a program execution sequence x_p, an interface I, and a

thread t, $\Pi_1(x_p)(t)(I)$ returns the sequence of controller actions in the order they appear in x_p. The recursive definition of this projection is similar to the one above. The only difference is that the sequence changes with the operations performed by t that are executions of actions defined in the controller whose interface is I instead of all the shared operations performed by t. Using this projection function, we define thread obedience as follows.

Definition 3.3.2 (Thread Obedience) *Given a thread t and a controller interface I t* **obeys** *I if for all execution sequences $x_p \in X_p$, $\Pi_1(x_p)(t)(I)$ is a legal sequences of I.*

Note that, thread obedience is weaker than the thread correctness since it considers only one controller interface.

Now we define the second projection function Π_2. Given a program execution $x_p = x_0, x_1, \ldots$ where $x_p \in X_p$ and a concurrency controller $CC = (\Gamma, IC, A, I)$, the function $\Pi_2(x_p)(CC)$ computes a projection of x_p on the concurrency controller CC. Let n be the maximum concurrent thread number in a program execution $x_p \in X_p$. Let also c_j be the program configuration that has the initial state of the controller CC. (I.e., c_j is the first configuration where there exists a store mapping that maps the concurrency controller to a value other then $\perp$ or uninitialized.) The projection $\Pi_2(x_p)(CC)$ is computed recursively as follows. $\Pi_2(x_0, ..., x_{j-1})(CC)$ is the empty sequence. $\Pi_2(x_0, \ldots, x_j)(CC)$= $\prod_{\gamma \in \Gamma} \ell_j^t(\gamma) \times \prod^n q_0$ where q_0 is the initial state of I, n is the maximum number of threads in the execution (Note that there is one interface state per thread), and t is the thread that has initialized the controller. The projection $\Pi_2(x_0, \ldots, x_j, \ldots, x_{l+1})(CC)$ is $\Pi_2(x_0, \ldots, x_j, \ldots, x_l)(CC)$ if x_{l+1} is

not $c_{l+1} \xrightarrow{sop^t(a)} c_{l+2}$ for some $t \in T$ where sop is an execution of an action of CC. (Recall that, the controller variables of CC are private and can only be accessed or modified with the actions of CC. Therefore, we only consider the operations that are an execution of an action of CC.) Otherwise, it is computed, assuming that the threads obey I, as follows. Let *act* $\in A$ be the action that is performed by t with the operation $sop^t(a)$. Let also, for each concurrent thread $t_z \in T$ and $t_z \neq t$, q_{t_z} be the interface state of t_z at the last element of the sequence $\Pi_2(x_0, \ldots, x_j, \ldots, x_l)(CC)$, and let q_t be that of t. The next interface state for each thread t_z other than t is $q'_{t_z} = q_{t_z}$. The next interface state (q'_t) of t is computed from the transition $(q_t, act, q') \in \delta$ and $q'_t = q'$. We can determine the next interface state q'_t since δ is deterministic (see Section 3.1). Finally, $\Pi_2(x_0, \ldots, x_j, \ldots, x_{l+1})(CC)$ is $\Pi_2(x_0, \ldots, x_j, \ldots, x_l)(CC)$, $(\prod_{\gamma \in \Gamma} \rho_{l+1}(\gamma) \times q'_t \times \prod_{t \in T \wedge t_z \neq t} q'_{t_z})$.

The condition that all threads obey to I is required since the computation of Π_2 relies on this assumption to construct such a sequence. If one of the threads violates this condition, the projection computation cannot find a next interface state for that thread at the violation point and cannot construct such a sequence.

Theorem 3.3.3 *Let X_p be the set of all executions of P. Given a concurrency controller $CC = (\Gamma, IC, A, I)$, if all the threads that use CC obey $I = (Q, q_0, V, \Sigma, \delta, F)$ then $\bigcup_{x_p \in X_p} \{\Pi_2(x_p)(CC)\}$ is the subset of the set of execution paths in $T(CC)(n)$, where n is the maximum number of threads.*

The proof relies on the constructive definition of the projection function Π_2. Recall that at any execution of P, the controller variables ($\gamma \in \Gamma$) are accessed or

modified only by the actions of CC. These actions are the shared operations, which are considered during the projection computation. Also, if all threads obey I, based on Definition 3.3.1, the access sequence to CC by each thread is a legal sequence of I. The interface I specifies the most general legal thread access to CC that are considered by the transition system $T(CC)(n)$. Therefore, there is a projection for each execution sequence and any two consecutive elements of the projection result is a tuple in RT. The proof is as follows.

To show the subset relation, we have to show that for any $x_p \in X_p$ $\Pi_2(x_p)(CC)$ is an execution path of $T(CC)(n)$ if all threads obey I. For this purpose, we need to show that if all the threads using CC obey I based on Definition 3.3.2, then 1) each element of the sequence produced by $\Pi_2(x_p)(CC)$ is an element of ST, and 2) for any two consecutive elements of the sequence $\Pi_2(x_p)(CC)$, say p_i and p_{i+1}, there is a tuple $(s_i, s_{i+1}) \in RT$.

The first condition holds trivially by the definition of the projection function since each element of the projection is in $\prod_{\gamma\in\Gamma} \text{DOM}(\gamma) \times \prod_n Q$ which is the definition of ST. Note that, the first element of the sequence resulting from $\Pi_2(x_p)(CC)$ is an element of IT due to the projection function definition. Also, in the projection result, the uncreated threads are at initial interface state, which is the case in IT, and the interface state of terminated threads do not change.

Now we examine the second condition. Let p_i, p_{i+1} be two consecutive elements of $\Pi_2(x_p)(CC)$. Consider the controller variables in p_i and p_{i+1}. Let sop^t be the shared operation performed by some thread t that is used during the computation of p_{i+1} and let ρ_i, ρ_{i+1} be the shared store mappings before and after the operation sop^t,

respectively. According to the definition of projection function, sop is an action of CC. In the following discussion, we denote this action as *act*. Let q_i^t be the interface state of t in p_i and q_{i+1}^t be the interface state of t in p_{i+1}. According to the definition of the projection function, the interface states of all other threads are the same in both p_i and p_{i+1}. If t obeys I, then according to Definition 3.3.1 there is a transition $(q_i^t, \mathit{act}, q_{i+1}^t) \in \delta$. Moreover, there is a tuple $(s_i, s_{i+1}) \in RT$ where $s_{i+1}(\gamma) = \rho_{i+1}(\gamma)$ and $s_i(\gamma) = \rho_i(\gamma)$ for all $\gamma \in \Gamma$, $s_{i+1}(Q)(t) = q_{i+1}^t$, $s_i(Q)(t) = q_i^t$, and $s_{i+1}(Q - t) = s_i(Q - t)$. Therefore, $(p_i, p_{i+1}) \in RT$ if t obeys I. ∎

Theorem 3.3.3 leads to the following.

Corollary 3.3.4 *Given a concurrency controller $CC = (\Gamma, IC, A, I)$ and a program P that has an instance of CC accessed by n threads, the ACTL properties verified on transition system $T(CC)(n)$ for the concurrency controller CC are preserved in P if all the threads are interface correct.*

To show that the ACTL properties of $T(CC)(n)$ are preserved in the program P we need to show that $T(CC)(n)$ simulates P [29]. For this purpose we define a simulation relation $H \subseteq \mathcal{C} \times ST$ where $\mathcal{C}$ is the configuration set of P and ST is the state set of $T(CC)(n)$. This relation is as follows. $H = \{(c, s) \mid c \in \mathcal{C} \wedge s \in ST \wedge (\exists x_p = x_0, x_1, \ldots, x_i, \cdots \in X_p \; s.t. \; x_i = c_i \xrightarrow{label} c \wedge \Pi_2(x_0, x_1, \ldots, x_i)(CC) = s_0, s_1, \ldots, s)\}$ where X_p is the set of executions of P. Such a simulation function exists if all the threads are interface correct since the set of projection Π_2 results for any $x_p \in X_p$ is a subset of the transition system $T(CC)(n)$ according to Theorem

3.3.3. ∎

Based on this corollary, the behavior verification of a concurrency controller is performed on the $T(CC)(n)$ in our framework.

3.3.3 Model for Distributed Programs

A distributed program is a collection of a number of single programs running on different machines. We classify distributed programs into two categories: 1) distributed programs DP whose participants communicate through remote procedure calls, 2) distributed programs DP_A whose participants communicate through asynchronous messaging. We assume that the number of participants is constant during the execution of distributed programs. In this section, we first introduce an abstract model for DP, and then continue with an abstract model for DP_A.

Program Model for Distributed Programs with Remote Method Invocations

A distributed program with remote method invocations $DP = (P_1, P_2, \ldots, P_k)$ is a tuple of single programs running on different machines (in Java different JVMs), where $k \geq 1$ is the number of single programs in DP. The model for a single program P_i in DP where $1 \leq i \leq k$ is the same as in Section 3.3.1. Due to the remote procedure calls, however, we need to add the following to the model of single program P_i participating DP: RMI threads, RMI operations, and RMI events.

In a distributed program DP, the runtime environment receiving the remote call creates implicit threads to serve these remote calls. The RMI specification [87, 81] states that "remote method invocation on the same remote object may execute con-

currently". We assume that the runtime environment creates an implicit thread for each RMI connection. Therefore, we add one implicit thread (called RMI thread t_{r_j}) per participant $P_j \in DP$ to the implicit threads of a participant $P_i \in DP$ where $1 \leq j \leq k$ and $j \neq i$.

A remote method invocation works as follows. Suppose a thread t of P_i invokes a remote method published by P_j. This request is delivered to the runtime environment of P_j and handled with an implicit thread t_{r_i}. During this process, the scheduler for P_j may schedule other threads of P_j, as discussed in the model for concurrent programs. In the mean time, the thread t stalls until a response from P_j is received. In this scenario we differentiate the invocation performed within P_i from the effect on P_j. We call the remote invocation an *RMI operation*, and the effect on the other side an *RMI event*. Below we define these operations and events.

The input events associated with P_j participating DP are the GUI events (see Section 3.3.1) and the RMI events. An RMI event influences the configuration of an RMI thread t_r. In our model, an RMI event affects the local store of t_r and later while handling the event, t_r reflects the effects on shared store with shared operations. We define an RMI event captured by t_r as $e_r(ea) : \mathit{Lcl}(t_r) \times \mathit{Ctl}(t_r) \rightarrow \mathit{Lcl}(t_r) \times \mathit{Ctl}(t_r)$ where ea is the attribute sequence of the RMI event.

RMI operations are one of the operation types a thread within P_i can perform. An RMI operation is a method invocation on another participant program of DP. We define an RMI operation performed by a thread t with an argument sequence a as $rop^t(a) : \mathit{Lcl}(t) \times \mathit{Ctl}(t) \rightarrow \mathit{Lcl}(t) \times \mathit{Ctl}(t)$. [1]

[1]The handler for the remote object might be in the shared store of P_i. In that case, the operations on this object are considered as shared operations. This is a similar approach to the one we used for

Let T^{P_i} be the finite set of threads at P_i and Sh^{P_i} be the set of shared store mappings at P_i. The set of configuration of P_i is defined as $\{c^{P_i} \mid c^{P_i} \in Sh^{P_i} \times \prod_{t \in T^{P_i}} (Lcl(t) \times Ctl(t))\}$. We denote this set as $\mathcal{C}^{P_i}$. Note that the thread set T^{P_i} includes the main thread, event thread, all the explicit threads, and the implicitly created RMI threads.

The initial configuration of P_i participating DP is $c_0^{P_i} = (\rho_0, \ell_0^{t_m}, \alpha_0^{t_m}, \ell_0^{t_e}, \alpha_0^{t_e}, \ell_0^{t_{r_0}}, \alpha_0^{t_{r_0}}, \ell_0^{t_{r_1}}, \alpha_0^{t_{r_1}}, \ldots, \ell_0^{t_{r_k}}, \alpha_0^{t_{r_k}})$. Here $\rho_0 \in Sh^{P_i}$ is the initial shared store mapping of P_i, $\ell_0^{t_m} \in Lcl(t_m)$ is the initial local store mapping of the P_i's main thread, $\ell_0^{t_e} \in Lcl(t_e)$ is the initial local store mapping of the P_i's event thread. For $1 \leq j \leq k$, $\ell_0^{t_{r_j}} \in Lcl(t_{r_j})$ is the initial local store mapping of the RMI thread serving calls initiated by P_j and targeted to P_i. Finally, $\alpha_0^{t_m} \in Ctl(t_m)$, $\alpha_0^{t_e} \in Ctl(t_e)$, and $\alpha_0^{t_{r_j}} \in Ctl(t_{r_j})$ are the initial control states of the main thread, event thread, and RMI threads, respectively.

Before defining executions of P_i, we define transition relations for RMI operations and RMI events. The relation for execution of an RMI operation $rop(a)$ with argument sequence a is as follows.

$$\begin{aligned} R_{rop^t}(a) = \; & \{(c, c') \mid c, c' \in \mathcal{C}^{P_i} \wedge rop^t(a)(c(Lcl(t)), c(Ctl(t))) = (\ell, \alpha) \\ & \wedge c'(Sh) = c(Sh) \wedge c'(Lcl(t)) = \ell \wedge c'(Ctl(t)) = \alpha \\ & \wedge (\forall t' \in c(T^{P_i}), t' \neq t \Rightarrow c'(Lcl(t')) = c(Lcl(t')) \\ & \wedge c'(Ctl(t')) = c(Ctl(t'))\} \end{aligned}$$

The relation for an RMI event with an attribute sequence ea captured with an

environment operations.

RMI thread t_r within P_i is

$$\begin{aligned} R_{e_r}(ea) = \quad & \{(c, c') \mid c, c' \in \mathcal{C}^{P_i} \ \wedge \ e_r(a)(c(\mathit{Lcl}(t_r)), c(\mathit{Ctl}(t_r))) = (\ell^{t_r}, \alpha^{t_r}) \\ & \wedge c'(\mathit{Sh}) = c(\mathit{Sh}) \ \wedge \ c'(\mathit{Lcl}(t_r)) = \ell^{t_r} \ \wedge \ c'(\mathit{Ctl}(t_r)) = \alpha^{t_r} \\ & \wedge (\forall t' \in c(T^{P_i}), t' \neq t_r \Rightarrow c'(\mathit{Lcl}(t')) = c(\mathit{Lcl}(t')) \\ & \wedge c'(\mathit{Ctl}(t')) = c(\mathit{Ctl}(t'))\} \end{aligned}$$

In the program P_i, transition from one configuration to another is performed according to the following relation.

$$R^{P_i} = \bigcup_{op^t(a) \in OP^t, t \in T^{P_i}} R_{op^t(a)} \cup \bigcup R_{e(ea)}$$

where the set of events E includes RMI events for all attribute sequences as well as the GUI events discussed earlier, and the set of operations OP^t performed by $t \in T^{P_i}$ includes RMI operations performed by t with any argument sequence as well as the operations discussed in Section 3.3.1. The execution of P_i is defined similar to the execution of a single program in Section 3.3.1.

The set of configuration of $DP = (P_1, P_2, \ldots, P_k)$ is $\{c \mid c \in \prod_{1 \leq i \leq k} \mathcal{C}^{P_i}\}$. The initial configuration of DP is $c_0 = \prod_{1 \leq i \leq k} c_0^{P_i}$ where $c_0^{P_i}$ is the initial configuration of P_i. The transition from one configuration of DP to another is defined with the following relation.

$$R^{DP} = \bigcup_{1 \leq i \leq k} R^{P_i}$$

An execution of DP starts from the initial configuration c_0 defined above. During the execution, transitions are performed according to the R^{DP}. A program execution is a sequence $x_{DP} = x_0, x_1, \ldots, x_i, x_{i+1}, \ldots$ where $x_i = c_i \xrightarrow{label} c_{i+1}$ with $(c_i, c_{i+1}) \in R^{DP}$. Here $label$ is either an operation performed by one of the threads

in one of the participating programs, or an input event handled by t_e or t_r in one of the participating programs.

Program Model for Distributed Programs with Asynchronous Messaging

A distributed program DP_A whose participants communicate through asynchronous messaging is a tuple $DP_A = (P_1, \Theta_1, \ldots, P_k, \Theta_k, class(M))$. where k is the number of participants, $class(M)$ is the finite set of message classes, P_i for $1 \leq i \leq k$ is a participant program same as in DP, and Θ_i is the configuration set of the input message queue of P_i.

The set of configurations of $DP_A = (P_1, \Theta_1, \ldots, P_k, \Theta_k, class(M))$ is $\{c \mid c \in \prod_{1 \leq i \leq k}(\mathcal{C}^{P_i} \times \Theta_i)\}$. We denote this set as $\mathcal{C}^{DP_A}$. We introduce two notations. We use $c(P_i)$ to denote the local configuration of P_i at a global configuration $c \in \mathcal{C}^{DP_A}$. We use $c(\Theta_i)$ to denote the value of the input queue of P_i at a global configuration $c \in \mathcal{C}^{DP_A}$.

The initial configuration of DP_A is $c_0 = \prod_{1 \leq i \leq k}(c_0^{P_i} \times \langle\rangle)$ where $c_0^{P_i}$ is the initial configuration of P_i and $\langle\rangle$ is the empty queue.

To define asynchronous messaging among the participant programs, we introduce two new kinds of operations: asynchronous message send operations (*comS*) and asynchronous message receive operation (*comR*). We define these operations as follows. Let P_i be the participant of DP_A sending a message instance $m \in M$ where $sender(m) = P_i$. This is performed with an asynchronous send operation $comS^{P_i}(m) : \mathcal{C}^{P_i} \times \Theta_j \rightarrow \mathcal{C}^{P_i} \times \Theta_j$ where Θ_j is the configuration set of the input message queue of P_j and $receiver(m) = P_j$. We define an asynchronous message re-

ceive operation performed by the participant program P_j as $\mathit{comR}^{P_j}(a) : \mathcal{C}^{P_j} \times \Theta_j \rightarrow \mathcal{C}^{P_j} \times \Theta_j$ where $a = a^0, a^1, \ldots, a^l$ is the argument sequence of the operation and the argument $a^0 \in M$ is a message instance.

The transition from a configuration of DP_A to another when a participant program P_i performs an asynchronous send operation with message m is defined with the following relation.

$$\begin{aligned} R_{\mathit{comS}^{P_i}(m)} = \; & \{(c, c') \mid c, c' \in \mathcal{C}^{DP_A} \; \wedge \; (\exists 1 \leq j \leq k, \; s.t. \\ & \mathit{comS}^{P_i}(m)(c(P_i), c(\Theta_j)) = (c^{P_i}, \mathit{append}(c(\Theta_j), \langle m \rangle)) \\ & \wedge c'(P_i) = c^{P_i} \; \wedge \; (\forall 1 \leq z \leq k, z \neq i \Rightarrow c'(P_z) = c(P_z)) \\ & \wedge c'(\Theta_j) = \mathit{append}(c(\Theta_j), \langle m \rangle) \\ & \wedge (\forall 1 \leq z \leq k, z \neq j \Rightarrow c'(\Theta_z) = c(\Theta_z)))\} \end{aligned}$$

The transition from a configuration of DP_A to another when a participant program P_j performs an asynchronous receive operation with an argument sequence a is defined with the following relation.

$$\begin{aligned} R_{\mathit{comR}^{P_j}(a)} = \; & \{(c, c') \mid c, c' \in \mathcal{C}^{DP_A} \; \wedge \; \mathit{first}(c(\Theta_j)) = m \\ & \wedge \mathit{comR}^{P_j}(c(P_j), c(\Theta_j)) = (c^{P_j}, \Theta) \; \wedge \; c(\Theta_j) = \mathit{append}(\Theta, \langle m \rangle)) \\ & \wedge c'(P_j) = c^{P_j} \; \wedge c'(\Theta_j) = \Theta \\ & \wedge (\forall 1 \leq z \leq k, z \neq j \Rightarrow c'(P_z) = c(P_z) \; \wedge \; c'(\Theta_z) = c(\Theta_z)))\} \end{aligned}$$

Finally, we define a relation R^{DP_A} as follows.

$$R^{DP_A} = R_{\mathit{comR}} \cup R_{\mathit{comS}} \cup \bigcup_{1 \leq i \leq k} R^{P_i}$$

where R^{P_i} is the same as in DP except that the definitions of the R^{P_i} relations have to be extended to preserve the state of the message queues.

We define an execution of DP_A as follows. The execution starts from the initial configuration c_0. The transition from one configuration to another is performed according to R^{DP_A}. We denote an execution as a sequence $x_{DP_A} = x_0, x_1, \ldots, x_i, x_{i+1}, \ldots$ where $x_i = c_i \xrightarrow{label} c_{i+1}$ and $(c_i, c_{i+1}) \in R_{DP_A}$. In this definition $label$ is one of the followings: an asynchronous send operation, an asynchronous receive operation, another kind of operation such as a local operation performed by a thread of one participant program, and an input event to a participant program. We denote the set of all program execution sequences as X_{DP_A}.

We formalize an asynchronously communicating composite web service as a DP_A. In our approach, such web services are implemented based on the peer controller pattern. According to this pattern, each participant program has *only* one peer controller. (We will use the terms participant program, participant peer, and peer interchangeably.) On the other hand, a peer program may have more than one concurrency controller. However, a shared data object is protected only by one concurrency controller. In short, there might be more than one concurrency controller in a peer program, but there is only one peer controller in a peer program.

3.3.4 Projections on Distributed Programs

In this section we define two projection functions so that we can form the basis for interface and behavior verifications for composite web services which are distributed programs communicating with asynchronous messaging (DP_A).

Given a program $DP_A = (P_1, \Theta_1, \ldots, P_k, \Theta_k, class(M))$ and an execution of this program $x_{DP_A} = x_0, x_1, \ldots$ where $x_{DP_A} \in X_{DP_A}$, the projection func-

tion $\Pi_3(x_{DP_A})$ is defined as follows. Let c_j be the first program configuration from which a participant program performs an asynchronous send operation with any message instance (i.e., the operation at $c_j \xrightarrow{comS_0(m_0)} c_{j+1}$ where $class(m_0) \in class(M)$ is the first asynchronous send operation performed by any of the participant programs.). $\Pi_3(x_0, ..., x_{j-1})$ is the empty sequence. $\Pi_3(x_0, \ldots, x_j) = m_0$ where m_0 is the message instance in the operation $comS_0(m_0)$. The projection $\Pi_3(x_0, \ldots, x_j, \ldots, x_{l+1}) = \Pi_3(x_0, \ldots, x_j, \ldots, x_l)$ if x_{l+1} is not $c_{l+1} \xrightarrow{comS^{P_i}(m)} c_{l+2}$ for some participant P_i where $class(m) \in class(M)$. Otherwise, the projection $\Pi_3(x_0, \ldots, x_j, \ldots, x_{l+1}) = \Pi_3(x_0, \ldots, x_j, \ldots, x_l), m$ where $class(m) \in class(M)$ is the message instance used in the operation $comS^{P_i}(m)$ performed by the participant P_i.

The output of a projection $\Pi_3(x_{DP_A})$ for an execution sequence $x_{DP_A} \in X_{DP_A}$ is a sequence of message instances in the order they are sent. Note that, this sequence is the *conversation* generated by the execution x_{DP_A}. Let $W = (class(M), I_1, I_2, \ldots, I_k)$ be the composite web service specification for DP_A where I_i is the peer interface of participant P_i. Below we will define the obedience of a participant to its peer interface, and show that the projection Π_3 of any execution sequence of DP_A is a conversation generated by the transition system $T(W)$ if the participants obey their peer interfaces.

To define the obedience to a peer interface, we first introduce another projection function Π_4. Given a program execution $x_{DP_A} = x_0, x_1, \ldots$ where $x_{DP_A} \in X_{DP_A}$, a program P participating DP_A, and a peer interface I, the projection $\Pi_4(x_{DP_A})(P)(I)$ is defined recursively as follows. Let c_j be the first program configuration from

which P performs an asynchronous send operation with any message instance or an asynchronous receive operation. The projection $\Pi_4(x_0, \ldots, x_j)(P)(I)$ is the empty sequence. For all $l \geq j$, the projection is computed recursively as follows. The projection $\Pi_4(x_0, \ldots, x_j, \ldots, x_{l+1}) = \Pi_4(x_0, \ldots, x_j, \ldots, x_l), ?m$ if $x_{l+1} = c_{l+1} \xrightarrow{comR^P(a)} c_{l+2}$ where $a^0 = m$ and $class(m) \in class(M)$. The projection $\Pi_4(x_0, \ldots, x_j, \ldots, x_{l+1}) = \Pi_4(x_0, \ldots, x_j, \ldots, x_l), !m$ if $x_{l+1} = c_{l+1} \xrightarrow{comS^P(m)} c_{l+2}$ where $a^0 = m$ and $class(m) \in class(M)$. Otherwise, the projection is $\Pi_4(x_0, \ldots, x_j, \ldots, x_{l+1}) = \Pi_4(x_0, \ldots, x_j, \ldots, x_l)$.

Definition 3.3.5 (Peer Obedience) *Given a peer program P participating DP_A and a peer interface I, P* **obeys** *I if for all execution paths $x_{DP_A} \in X_{DP_A}$ the projection $\Pi_4(x_{DP_A})(P)(I)$ is a legal sequences of I.*

Based on Π_3 definition above and the obedience in Definition 3.3.5, we give the following theorem.

Theorem 3.3.6 *Let $DP_A = (P_1, \Theta_1, P_2, \Theta_2, \ldots, P_k, \Theta_k, class(M))$ be a distributed program with asynchronous messaging and let X_{DP_A} be the set of all executions of DP_A. Given a composite web service specification $W = (class(M), I_1, I_2, \ldots, I_k)$ for DP_A, $\bigcup_{x_{DP_A} \in X_{DP_A}} \{\Pi_3(x_{DP_A})\}$ is the subset of the conversation set generated by $T(W) = (IT, ST, RT)$ if for all $1 \leq i \leq k$ the participant P_i obeys I_i.*

First, since the set of message classes are the same in DP and W, the elements of the projection result are the elements of the $T(W)$'s conversations.

Now, we need to show that the projection $\Pi_3(x_{DP_A})$ for each $x_{DP_A} \in X_{DP_A}$ is a conversation generated by $T(W)$ if all the participant peers obey their peer

interface. Let con be the result of $\Pi_3(x_{DP_A})$ for an execution sequence $x_{DP_A} \in X_{DP_A}$. Since the participant peers obey their interfaces, by Definition 3.3.5, these peers send or receive messages in an order specified by their interfaces. Recall that, the transition system $T(W)$ includes all possible message exchange orderings of these interfaces. Therefore, there is an execution path $s_0, s_1, \ldots$ of $T(W)$ that corresponds to x_{DP_A}. The conversation generated by this execution path is the same as con since the projection considers only the sent messages and outputs a message sequence in the order they are sent. Therefore, con is an element of the conversation set of $T(W)$.

∎

3.4 Thread Isolation

In this section we present our thread isolation techniques so that we can perform the interface verification for each thread separately. Given a concurrent thread t in a program, we denote the isolated program for t as P'. Since there is only one thread, a configuration of P' has only the control state of t and the local store mapping of t in addition to shared store mapping, i.e., $\mathcal{C}^{P'} = \{c|Sh \times c \in Lcl(t) \times Ctl(t)\}$. In this section, we define how we construct the isolated program P' in detail.

We isolate a thread by modeling the interactions with its environment and with other threads by using stubs, drivers and controller interfaces. Below we explain how we model these interactions and show how we generate an isolated program for Java threads.

The isolation techniques introduce nondeterminism. We define a function *choose* to be used for modeling nondeterminism in the rest of this section. The function *choose* takes a set as argument and returns one element of the set nondeterministically.

3.4.1 Modeling Environment Interaction Operations with Stubs

One form of thread-environment interaction is through environment interaction operations (eop). As discussed in Section 3.3.1 and 3.3.3, there are four types of such operations in a distributed program DP: GUI operations, file read and write operations, socket operations, and RMI operations. We use stubs to model each of these environment interaction operation types. A stub for an operation type abstracts the effect of the environment through that operation while conservatively preserving the influences on the thread execution with respect to the interface verification correctness criteria.

Let $eop^t(a)$ be an environment interaction operation performed by a thread t with the argument sequence a where a^0 holds the return value of the method call and a^1 holds the exceptions thrown during the operation. This operation (which is a method invocation) alters the state of the target objects, updates the control state of t, and may influence the rest of the execution of t with the return value of the method call and the exceptions thrown during the method call. While modeling such operations we do not store the state of the target objects and use nondeterminism instead. We use stubs to model such operations. The stub ($stub(a)$) produces all possible return values in $\text{DOM}(a^0)$ and throws all combinations of possible exceptions in $\text{DOM}(a^1)$.

With these stubs, the effects of the environment are preserved. Given an environment interaction operation eop^t performed by t, the stub of this operation is defined as $stub(a) : \mathit{Lcl}(t) \times \mathit{Ctl}(t) \rightarrow \mathcal{P}(\mathit{Lcl}(t) \times \mathit{Ctl}(t))$. The range is a power set since there is a thread configuration for each value returned from the method invocation and for each exception thrown (or not thrown). The possible control states ($\alpha \in \mathit{Ctl}(t)$) are the program point following the method call and the exception handling point.

Here we define one step execution of an isolated program P' with respect to t from a configuration c with $stub(a)$ where a is the argument sequence. We define the following relation

$$\begin{aligned} R_{stub(a)} = \quad & \{(c, c') | c, c' \in \mathcal{C}^{P'} \ \wedge \mathit{choose}(stub(a)(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t)))) = (\ell, \alpha) \\ & \wedge c'(\mathit{Lcl}(t)) = \ell \ \wedge c'(\mathit{Ctl}(t)) = \alpha \ \wedge \ c'(\mathit{Sh}) = c(\mathit{Sh})\} \end{aligned}$$

This relation defines the transitions from a configuration to another with a $stub(a)$. We denote one step execution from c with $stub(a)$ with $c \xrightarrow{eop(a)} c'$ where $(c, c') \in R_{stub(a)}$ and $eop(a)$ is the operation replaced by $stub(a)$. Note that, because of the nondeterminism (*choose*), there are a number of executions for an isolated program P' even though there is only one thread and one possible thread scheduling.

We realize these abstractions for environment interaction operations for Java program as follows. First we need to identify such operations in a Java program statically. RMI operations are the invocations of the methods of a remote object. Therefore, the identification of remote operations becomes identification of remote objects and their methods. In Java, the remote objects are instances of the classes that implements `java.rmi.Remote`. The invocations of remote object binding and look up methods are also considered as RMI operations. Unlike the first ones, the

latter methods are predetermined; they are the methods of `java.rmi` library.

The GUI operations are predetermined. For example, in Java, the methods for GUI operations are in the graphical libraries such as `java.awt` and `javax.swing`. (We assume that any graphical method outside the graphical library eventually reaches a method within the library. If this assumption does not hold, then such methods should be identified as well.) Similarly, the methods for file operations are within the `java.io`.

The socket operations are predetermined as well. Here we explain how socket operations are performed in Java. There are two types of communication protocols: TCP and UDP. Java provides a `java.net.Socket` class for TCP communications and a `java.net.DatagramSocket` class for UDP communications. For TCP communications, a program reads data from a `Socket` as a stream through a `java.io.Reader` object. (A typical Java program reads this stream through an object of `BufferedReader` class, which is a subclass of `Reader`.) Sending data is performed through an `OutputStream` object associated with the `Socket`. For UDP communications, programs read packets from a `DatagramSocket` via a `DatagramPacket`. Sending data is performed via `DatagramSocket` objects.

The next step is the abstraction via replacement of these operations with stubs. For Java programs, we achieve these abstractions by stub class substitutions. A stub of class contains every accessible method declaration of that class. The invocation of a method in a stub class by t realizes the $stub$ explained above. We implemented *choose* with the JPF's nondeterminism utilities `Verify.random(int)` and `Verify.randomBool()`. The nondeterminism utilities force JPF to search exhaus-

tively for every possible choice. Therefore, at verification time, the nondeterminism in the code results in an exhaustive search, not in random testing.

As an example consider a GUI operation with an invocation of the method `isRowSelected` of a table object (of class `javax.swing.JTable`). The stub of this operation should produce every possible thread configurations. We realize this stub operation with a call to the following method of the stub class for the table.

```
public boolean isRowSelected(int row)
               throws IllegalArgumentException{
   if(Verify.randomBool()) throw new IllegalArgumentException();
   return Verify.randomBool();
}
```

We have developed generic stub classes for the file, socket, and GUI operations since these operations are predetermined. For RMI operations we automatically generate stubs. Our generator inspects the remote interface to collect the RMI operations, and then synthesizes a stub class for the RMI objects. In addition to these operation stubs, since JPF is only able to handle pure Java, we also replace all calls to native code with pre-implemented stubs.

3.4.2 Modeling Shared and Asynchronous Communication Operations with Interfaces

In a program written based on the two DFV patterns, the shared operations and asynchronous communication operations are explicit. These operations are known beforehand. As discussed in Section 3.3, the shared operations are the invocations of the methods of shared objects which are either concurrency controller instances

or shared data objects protected by a concurrency controller. With controller interfaces and shared data stubs we can model the shared operations performed by a thread t and therefore abstract the threads with whom t interacts. The asynchronous communication operations are known at verification time as well. These operations are the invocations of peer controller actions. With peer interfaces we can model the asynchronous communication operations performed by a peer P and therefore abstract the other peers with whom P interacts. Below we first discuss modeling shared operations, and then continue with modeling asynchronous communication operations.

One category of shared operations is concurrency controller action invocations. We abstract the controller action invocations with the corresponding interface method invocations ($si(a)$). For example, the call to `r_enter` action of `BBRWController` is abstracted with the call to `r_enter` method of `BBRWStateMachine` (see Figures 2.3 and 2.7). The definition of an interface method invocation is as follows. Let $CC = (\Gamma, IC, A, I)$ be a concurrency controller with $I = (Q, q_0, V, \Sigma, \delta, F)$ and let *act* $\in A$ be the action associated with $sop(a)$. We define $si(a) : \mathit{Sh} \times \mathit{Ctl}(t) \rightarrow \mathit{Sh} \times \mathit{Ctl}(t) \cup \{\mathit{Error}\}$ for $sop(a)$ as

$$\begin{aligned} si(a) \equiv \;& \texttt{assert}(\exists q \in Q,\ s.t.\ (\texttt{cur}, \mathit{act}, q) \in \delta) \\ & \wedge \texttt{cur} := q \ \wedge\ (\texttt{cur}, \mathit{act}, q) \in \delta \ \wedge q \in Q \end{aligned}$$

where $\texttt{cur} \in Q$ denotes the current interface state t is in. Given a program configuration c, the value of `cur` in c is $c(\mathit{Sh}(\texttt{cur}))$. When the assertion fails the function $si(a)$ returns *Error*. We define the transition from one configuration to another with

$si(a)$ as

$$\begin{aligned} R_{si(a)} = \quad & \{(c, c') | c, c' \in \mathcal{C}^{P'} \ \wedge \ (si(a)(c(\mathit{Sh}), c(\mathit{Ctl}(t))) = \mathit{Error} \ \wedge \ c' = c_E) \\ & \vee (si(a)(c(\mathit{Sh}), c(\mathit{Ctl}(t))) = (\rho, \alpha) \ \wedge \ c'(\mathit{Sh}) = \rho \\ & \wedge c'(\mathit{Ctl}(t)) = \alpha) \ \wedge \ c'(\mathit{Lcl}(t)) = c(\mathit{Lcl}(t))\} \end{aligned}$$

where c_E denotes an error configuration. We represent one step execution from c with $si(a)$ modeling $sop(a)$ as $c \xrightarrow{sop(a)} c'$ where $(c, c') \in R_{si(a)}$.

The other category of shared operations is the shared data method invocations. We abstract such invocations with the corresponding data stub method invocations ($sstub(a)$). For example, the call to `insert` method of `DataBufferImpl` is abstracted with a call to `insert` method of `DataBufferStub`. We define a data stub method invocation as $sstub(a) : \mathit{Sh} \times \mathit{Ctl}(t) \rightarrow \mathcal{P}(\mathit{Sh} \times \mathit{Ctl}(t)) \cup \{\mathit{Error}\}$. Given a shared store mapping ρ and a control state α, the stub $sstub(a)(\rho, \alpha)$ for $sop^t(a)$ returns *Error* if the current interface state t is in is not a legal state to perform sop, which is defined in the interface specification. In other words, let $Q' \subseteq Q$ be the legal interfaces states to perform sop on a shared data. Then, $sstub(a)(\rho, \alpha)$ returns *Error* if $\rho(\texttt{cur}) \notin Q'$.

We define the transition from one configuration to another with $sstub(a)$ as

$$\begin{aligned} R_{sstub(a)} = \quad & \{(c, c') | c, c' \in \mathcal{C}^{P'} \ \wedge \ (sstub(a)(c(\mathit{Sh}), c(\mathit{Ctl}(t))) = \mathit{Error} \\ & \Rightarrow c' = c_E) \ \wedge \ (sstub(a)(c(\mathit{Sh}), c(\mathit{Ctl}(t))) \neq \mathit{Error} \\ & \Rightarrow \mathit{choose}(sstub(a)(c(\mathit{Sh}), c(\mathit{Ctl}(t)))) = (\rho, \alpha) \\ & \wedge c'(\mathit{Sh}) = \rho \ \wedge c'(\mathit{Ctl}(t)) = \alpha c'(\mathit{Lcl}(t)) = c(\mathit{Lcl}(t)))\} \end{aligned}$$

where c_E denotes an error configuration. We represent one step execution from c with $sstub(a)$ modeling $sop(a)$ as $c \xrightarrow{sop(a)} c'$ where $(c, c') \in R_{sstub(a)}$. Note that

since P' contains only the thread t, the variables appearing in $\rho \in$ *Sh* are accessed only by t.

In the concurrency controller pattern, the shared operations are encapsulated in the concurrency controller classes and shared data classes (shown as `Controller` and `Shared` in Figure 2.2). We realize the above shared operation abstractions by substituting controller interface machines for concurrency controllers and data stubs for the shared data protected by these controllers. In Figure 2.2, the data stub is displayed as `SharedStub` and the interface machine is displayed as `Controller-InterfaceMachine`. The methods of `ControllerInterfaceMachine` implement the $si(a)$, and the methods of `SharedStub` implements the $sstub(a)$ explained above. The function *choose* used in these definitions is realized with nondeterminism mechanism of JPF as discussed in the preceding section.

Controller interface machines and data stubs are local to t, i.e., they appear only in the local store mappings $\ell \in$ *Lcl*(t). By replacing concurrency controllers with interface machines and data objects with data stubs instances the shared objects are elided and modeled with local variables.

Similar to the shared operations, the asynchronous communication operations are explicit and known at verification time. These operations are the executions of peer controller actions. Given a composite web service $DP_A = (P_1, \Theta_1, \ldots, P_k, \Theta_k, class(M))$ we can isolate each participant program P_i for $1 \leq i \leq k$ by using their peer interfaces. With peer interfaces we can model the asynchronous communication operations (*comS* and *comR*) performed by a peer P_i and therefore abstract the other peers with whom P_i interacts.

Given an asynchronous send operation *comS* performed at P_i by an application thread [2] t with a message instance $m \in M$, we define $scomS(m)$: $Lcl(t) \times Ctl(t) \rightarrow Lcl(t) \times Ctl(t) \cup \{Error\}$ for this operation as follows. Let $I = (Q, q_0, V, \delta, F)$ be a peer interface for P_i and ℓ be the local store mapping when the operation is activated. Also let $\eta_0, \eta_1, \ldots, \eta_l$ be a sequence of values where for $1 \leq z \leq l$, $\eta_z = \ell(v_z)$ is the value of interface variable $v_z \in V$.

$$
\begin{aligned}
scomS(m) \equiv \; & \texttt{assert}(\exists q' \in Q,\ s.t.\ (\texttt{cur}, !class(m), g, u, q') \in \delta \\
& \wedge\ g(attr(m), \eta_0, \eta_1, \ldots, \eta_l))\ \wedge\ (\texttt{cur}, !class(m), g, u, q) \in \delta \\
& \wedge\ g(attr(m), \eta_0, \eta_1, \ldots, \eta_l))\ \wedge v'_0, v'_1, \ldots, v'_l := u(\eta_0, \eta_1, \ldots, \eta_l) \\
& \wedge\ \texttt{cur} := q\ \wedge\ q \in Q
\end{aligned}
$$

where $\texttt{cur} \in Q$ denotes the current peer interface state the application thread t is in and for $1 \leq z \leq l$, $v'_z \in V$. When the assertion fails, the function $scomS(m)$ returns *Error*. We define the transition from one configuration to another with $scomS(m)$ as

$$
\begin{aligned}
R_{scomS(m)} = \; & \{(c, c') | c, c' \in \mathcal{C}^{P'}\ \wedge\ (scomS(m)(c(Lcl(t)), c(Ctl(t))) = Error \\
& \wedge\ c' = c_E)\ \vee\ (scomS(m)(c(Lcl(t)), c(Ctl(t))) = (\ell, \alpha) \\
& \wedge\ c'(Lcl(t)) = \ell\ \wedge\ c'(Ctl(t)) = \alpha)\ \wedge\ c'(Sh) = c(Sh)\}
\end{aligned}
$$

where c_E denotes an error configuration. We represent one step execution from a configuration c with $scomS(m)$ modeling $comS(m)$ as $c \xrightarrow{comS(m)} c'$ where $(c, c') \in R_{scomS(m)}$.

Given an asynchronous receive operation *comR* performed at P_i with an argument sequence a, we define $scomR$: $Lcl(t) \times Ctl(t) \rightarrow \mathcal{P}(Lcl(t) \times Ctl(t)) \cup \{Error\}$

[2] According to the peer controller pattern, the application threads are session based. The application threads within a program P_i do not affect each other. Moreover, each application thread should obey the peer interface.

for this operation as follows. Let $I = (Q, q_0, V, \delta, F)$ be a peer interface for P_i.

$$\begin{aligned} \mathit{scomR}(a) \equiv\ & \texttt{assert}(\exists q' \in Q,\ s.t.\ (\texttt{cur}, ?class(m), g, u, q') \in \delta) \\ & \wedge\ (\texttt{cur}, ?class(m), g, u, q) \in \delta\ \wedge\ v'_0, v'_1, \ldots, v'_l := u(\eta_0, \eta_1, \ldots, \eta_l) \\ & \wedge\ a^0 := \mathit{choose}(\{m_i \mid class(m_i) = class(m)\}) \\ & \wedge\ \texttt{cur} := q\ \wedge\ q \in Q \end{aligned}$$

where $m \in M$, a^0 is the first element of a that represents the return value of the Java method implementing *scomR*, for $1 \leq z \leq l$, $v'_z \in V$, for $1 \leq z \leq l$, η_z is the value of the interface variable $v_z \in V$ mapped by the local store mapping at the configuration when the operation is activated, and $\texttt{cur} \in Q$ denotes the current peer interface state the application thread t is in. In this definition, a^0 is assigned to a message instance with arbitrary attribute values. We define the transition from one configuration to another with $\mathit{scomR}(a)$ as

$$\begin{aligned} R_{\mathit{scomR}(a)} = \ & \{(c, c') | c, c' \in \mathcal{C}^{P'}\ \wedge\ (\mathit{scomR}(a)(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = \mathit{Error} \\ & \wedge\ c' = c_E)\ \wedge\ (\mathit{scomR}(a)(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) \neq \mathit{Error} \\ & \Rightarrow \mathit{choose}(\mathit{scomR}(a)(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t)))) = (\ell, \alpha) \\ & \wedge\ c'(\mathit{Lcl}(t)) = \ell\ \wedge\ c'(\mathit{Ctl}(t)) = \alpha)\ \wedge\ c'(\mathit{Sh}) = c(\mathit{Sh})\} \end{aligned}$$

where c_E denotes an error configuration. We represent one step execution from a configuration c with $\mathit{scomR}(a)$ modeling $\mathit{comR}(a)$ as $c \xrightarrow{\mathit{scomR}(a)} c'$ where $(c, c') \in R_{\mathit{scomR}(a)}$.

These abstractions for asynchronous operations are realized by substituting `CommunicationInterface` for `CommunicationController` (see Figure 2.12). The methods of `CommunicationInterface` implement *scomS* and *scomR* via the fi-

nite state machine implementation `StateMachine` provided with the framework (see Section 2.2.4).

3.4.3 Modeling Thread Initialization and Input Events with Drivers

Drivers are necessary to transform a thread execution to a stand-alone program execution. As discussed in Section 3.3, there are three types of threads: explicit threads, implicit threads, and the main thread. We define different types of drivers for each of these categories.

The driver for an explicit thread t is responsible for setting the initial configuration of t and instantiating the thread. In other words, an explicit thread driver constructs the initial configuration $c_0 = (\ell_0, \alpha_0)$ of the isolated program P' for t.

For the implicit threads, a driver models the execution of the implicit thread by producing all possible input event sequences related to that kind of thread. Recall that there are two kinds of implicit threads (t_r and t_e) and two kinds of input events: the RMI events that are handled by an RMI thread t_r, and the GUI events that are handled by the event thread t_e. Let E be the set of available input events with any event attribute sequence at a program configuration c. The available RMI events are all RMI events unless there is a state machine specifying a legal RMI event sequence. The available GUI events are the events related to the GUI objects that are visible to the users of the program at the configuration c. We give the following relation to

define a transition from a configuration with an input event.

$$R_e = \{(c, c') | c, c' \in \mathcal{C}^{P'} \wedge \mathit{choose}(E)(c(\mathit{Lcl}(t)), c(\mathit{Ctl}(t))) = (\ell, \alpha) \\ \wedge c'(\mathit{Lcl}(t)) = \ell \wedge c'(\mathit{Ctl}(t)) = \alpha \wedge c'(\mathit{Sh}) = c(\mathit{Sh})\}$$

where t is either the event thread or an RMI thread. Note that, because of the nondeterminism with *choose*, a program isolated with such a driver for an implicit thread has a number of different execution sequences including the executions of the implicit thread at the original program. Therefore, we achieve a conservative abstraction of the implicit threads.

For the main thread, a driver models the environment interaction of command line argument read. Let eop be the command line argument read. In an execution sequence of P this operation appears at $x_0 : c_0 \xrightarrow{eop^{tm}(a)} c_1$ where a is the argument sequence and a^0 holds the values provided by the command line. (Actually, a^0 holds these values in an array.) Since this is an environment interaction operation, it is modeled with the corresponding $stub(a)$ discussed in Section 3.4.1. The driver of the main thread first executes the $stub(a)$ and then performs the operations of the main thread (by calling the `main` method).

Here we discuss how we realize these models with drivers for Java programs and how we generate drivers for each thread. The driver for the event thread t_e first launches the GUI components, and finds all the visible and enabled GUI objects that have registered event listeners. This is the initialization of the program, i.e, constructing the initial configuration of P'. After the initialization, the driver enters a loop generating the input event sequence. At each iteration of the loop, the driver first chooses one of these GUI objects, chooses an event, and then calls the listeners

for that event. In other words, at each iteration, the driver changes the program configuration as defined in the relation R_e above. We automatically generate an event thread driver and expect the user to perform data value assignments using the results of a data dependency analysis explained in the next section. During driver generation the GUI component launch mechanism is created by copying the relevant part from the application code, all possible user event types are identified by finding all different event listener types in the code.

The driver of an RMI thread t_r performs initialization and produces all possible input event sequences in a loop by calling every remote method with every possible value for the elements of the method's argument sequence. In other words, after initialization, the driver changes the program configuration according to R_e at each iteration of a loop. We automatically generate the driver for an RMI thread using the remote Java interfaces that define signatures of remote methods. The generator inspects these remote interfaces to collect the RMI events. The generator also examines the concrete class implementing the remote interface to synthesize a code for initializations. If the concrete class looks up another RMI component (i.e. if there is a call to `Naming.lookup(String)` method), the generator creates the code for registering the RMI stub of corresponding component to the stub of the `Naming` class. Then, the generator puts an instantiation of the given concrete class into the driver code. After the generation of the driver, the user can modify the value assignments depending on the results of a dependency analysis which will be discussed in Section 3.4.4.

The driver for a main thread is straightforward. The driver chooses one of the

possible values for command line arguments and invokes the `main` method. We generate a skeleton for such drivers and expect the user to alter the value assignments. To generate a driver for an explicit thread, our generator inspects the constructor of the thread. Then, it synthesizes the code to create the constructor arguments and to initiate the thread execution. This code is a call to the thread constructor followed by a call to the thread's `run` method. The driver does not call the `start` method which in turn will call `run`, since it creates another thread in the JVM. To avoid the new thread creation, the driver calls `run` directly.

Modeling Thread Creation and Termination Operations

The explicit thread creation operation $tcop$ is modeled with a one-time-written stub. With this stub the effect of $tcop$ is reflected on the creator thread t. Since the isolated program P' for t has no other thread, the stub operation does not alter the shared store mapping. The effect of $tcop$ on the created thread t' is modeled in the isolated program for t' with the explicit thread driver discussed in the preceding section. The relation for the stub $stc : \mathit{Lcl}(t) \times \mathit{Ctl}(t) \rightarrow \mathit{Lcl}(t) \times \mathit{Ctl}(t)$ that is used by the isolated creator thread t is

$$\begin{aligned} R_{stc} = \quad & \{(c, c') | c, c' \in \mathcal{C}^{P'} \ \wedge \ stc(\mathit{Lcl}(t), \mathit{Ctl}(t)) = (\ell, \alpha) \\ & \wedge c'(\mathit{Lcl}(t)) = \ell \ \wedge c'(\mathit{Ctl}(t)) = \alpha \ \wedge \ c'(\mathit{Sh}) = c(\mathit{Sh})\} \end{aligned}$$

The thread creation operation is abstracted with `public void start()` that resides in the provided stub class for `java.lang.Thread`. This stub method does nothing. The effect of this method is that the caller thread t updates its control state.

Finally, the effect of thread termination in an isolated program is the termination

of the program since there are no other threads. Therefore, we do not abstract thread terminations and leave them as is.

3.4.4 Data Dependency Analysis

Some elements of the argument sequence of operations and events that are passed to a thread via drivers and stubs may not influence the synchronization behavior of the thread. These influences are the influences on the execution of shared operations sop that might lead to an interface violation. In other words, we need an analysis to determine data independence with respect to interface verification. We implemented a data dependency analysis to identify such elements of the argument sequence of all the events and operations that are modeled with the preceding isolation techniques.

The analysis is multiple backward traversals on the program dependence graphs [83]. The starting point of each traversal is determined as follows. For each method in the program, if there are branching statements that determine whether or not a concurrency controller or a shared data method is called, then each of these statements is a starting point of a backward traversal. These starting points are the statements that control the execution of a shared operation and computed using the control flow graph of the method. During the traversal the control and data dependency edges are followed backward and the visited definition sites are collected. The visited statements are marked to avoid entering infinite loops. The traversal should be interprocedural and capture the implicit dependencies between the methods of the same class. Such dependencies occur when one method uses the value of a class field and another method sets the value of that field. For example, a get and a set method of

the same class have such implicit dependency. The result of this procedure is a backward dependence tree per starting point whose vertices are the collected definition sites. The leaves are the influencing argument sequence elements and a path in the tree shows how these elements control the execution of shared operations.

We implemented this analysis using the Soot Java optimization framework [93], which uses a 3-address representation for Java. The analysis determines which statements, directly or indirectly, affect the reachability of invocation of a method that belongs to a concurrency controller or a shared data class. The analysis we implemented is context insensitive. In the implementation, instead of computing the program dependence graph, the control and data dependencies are computed on the fly. To capture the implicit dependencies, before the traversals, we compute for each class field the set of methods updating its value (and the value of its elements). The implemented on the fly backward traversal and definition site collection is a recursive backward dependence tree construction. The root of the tree is the current statement. At the beginning the current statement is the starting point statement. For each definition site of the variables in the statement and each branch site affecting the statement, we find the r-values used at those sites. If it is a local variable, we compute its backward dependency tree recursively and add this tree as a child. If it is a class field, we add the recursively computed backward dependency trees of each definition site of that class field as children. If it is a return value of another method, the recursively computed backward dependency tree of the callee's return statement is added as a child. If it is a parameter of the method, we find the call sites by using the call graph of the program and add their recursively computed dependency trees

as children.

The analysis results are used in the construction of drivers and stubs. If an element of the argument sequence a of an operation implemented in a stub does not influence the synchronization behavior, a constant value with the correct type is given to it in the stub implementation. Similarly, if an element of ea for an input event e does not influence the synchronization behavior, a constant value with the correct type is given to it when constructing the event e in the driver implementation. For the elements that might influence the synchronization behavior there are two possibilities. If the domain of such a value is finite (e.g. boolean) we enumerate all possible values and choose one value using JPF's nondeterminism utilities. Otherwise, the analysis results are inspected and necessary values are provided by the user.

3.4.5 Discussion on Isolation Techniques

In this section we show that the interface verification can be performed on the isolated programs for each thread separately based on our isolation technique. We first give the execution definition for a single thread. Then we give the properties of isolated programs and give a theorem that enables modular interface verification of a program.

Execution and Configuration of a Thread

The configuration of t is $c^t \in \mathit{Sh} \times \mathit{Lcl}(t) \times \mathit{Ctl}(t)$. We denote this set of configurations as $\mathcal{C}^t$. An execution of a general thread is a sequence of thread configurations

and operations performed by that thread. We define the following relation.

$$
\begin{aligned}
R^t_{op} = \quad & \{(c^t_i, c^t_j) \in \mathcal{C}^t \ \wedge (\exists sop^t(a)(c^t_i(\mathit{Sh}), c^t_i(\mathit{Ctl}(t))) = (\rho, \alpha^t) \\
& \quad \wedge c^t_j(\mathit{Sh}) = \rho \ \wedge c^t_j(\mathit{Lcl}(t)) = c^t_i(\mathit{Lcl}(t)) \ \wedge c^t_j(\mathit{Ctl}(t)) = \alpha^t) \\
& \vee (\exists lop^t(a)(c^t_i(\mathit{Lcl}(t)), c^t_i(\mathit{Ctl}(t))) = (\ell^t, \alpha^t) \\
& \quad \wedge c^t_j(\mathit{Sh}) = c^t_i(\mathit{Sh}) \ \wedge c^t_j(\mathit{Lcl}(t)) = \ell^t \ \wedge c^t_j(\mathit{Ctl}(t)) = \alpha^t) \\
& \vee (\exists eop^t(a)(c^t_i(\mathit{Lcl}(t)), c^t_i(\mathit{Ctl}(t))) = (\ell^t, \alpha^t) \\
& \quad \wedge c^t_j(\mathit{Sh}) = c^t_i(\mathit{Sh}) \ \wedge c^t_j(\mathit{Lcl}(t)) = \ell^t \ \wedge c^t_j(\mathit{Ctl}(t)) = \alpha^t) \\
& \vee (\exists tcop^t(a)(c^t_i(\mathit{Sh}), c^t_i(\mathit{Lcl}(t)), c^t_i(\mathit{Ctl}(t))) = (\rho, \ell^t, \alpha^t, \ell^{t'}, \alpha^{t'}) \\
& \quad \wedge c^t_j(\mathit{Sh}) = \rho \ \wedge c^t_j(\mathit{Lcl}(t)) = \ell^t \ \wedge c^t_j(\mathit{Ctl}(t)) = \alpha^t)\}
\end{aligned}
$$

During an execution of a general thread t, the transition from a configuration to another is based on the relation $R^t = R^t_{op}$.

As noted in the beginning of this section, there are two kinds of special threads: the main thread and the implicitly created thread. The main thread (t_m) is created when the program starts. The execution of the main thread begins with an environment interaction: reading the command line arguments. The rest of the execution is the same as a general thread execution.

One implicitly created thread is the event thread t_e. Event thread is created by the runtime environment (e.g. JRE) if there are visible GUI components in the program. For simplicity, we assume that t_e is created at the initial configuration of the program P. Whenever a GUI event occurs, t_e processes that input event and executes the event handlers that are implemented by the developers. These event handlers are

can contain local operations or shared operations. We define the following relation.

$$R_e^{t_e} = \{(c_i^{t_e}, c_j^{t_e}) | c_i^{t_e}, c_j^{t_e} \in \mathcal{C}^{t_e} \ \wedge (\exists e(ea)(c_i^{t_e}(\mathit{Lcl}(t_e)), c_i^{t_e}(\mathit{Ctl}(t_e))) = (\ell^{t_e}, \alpha^{t_e}) \\ \wedge c_j^{t_e}(\mathit{Sh}) = c_i^{t_e}(\mathit{Sh}) \ \wedge c_j^{t_e}(\mathit{Lcl}(t_e)) = \ell^{t_e} \ \wedge c_j^{t_e}(\mathit{Ctl}(t_e)) = \alpha^{t_e})$$

where e is a GUI event. The configuration of an event thread t_e is updated during an execution according to the relation $R^{t_e} = R_{op}^{t_e} \cup R_e^{t_e}$.

The other implicitly created threads are the RMI threads. RMI threads are created by the runtime environment to serve the remote calls triggered by other programs. We define the following relation.

$$R_e^{t_r} = \{(c_i^{t_r}, c_j^{t_r}) | c_i^{t_r}, c_j^{t_r} \in \mathcal{C}^{t_r} \ \wedge (\exists e(ea)(c_i^{t_r}(\mathit{Lcl}(t_r)), c_i^{t_r}(\mathit{Ctl}(t_r))) = (\ell^{t_r}, \alpha^{t_r}) \\ \wedge c_j^{t_e}(\mathit{Sh}) = c_i^{t_r}(\mathit{Sh}) \ \wedge c_j^{t_e}(\mathit{Lcl}(t_r)) = \ell^{t_r} \ \wedge c_j^{t_e}(\mathit{Ctl}(t_r)) = \alpha^{t_r})$$

where e is an RMI event. The configuration of an RMI thread t_r is updated during an execution according to the relation $R^{t_r} = R_{op}^{t_r} \cup R_e^{t_r}$.

An execution sequence of thread t is represented as $x^t = x_0, x_1, \ldots, x_i, x_{i+1}, \ldots$ where $x_i = c_i^t \xrightarrow{label} c_{i+1}^t$ with $(c_i^t, c_{i+1}^t) \in R^t$ and $label$ is an operation $op^t(a)$ with an argument sequence a performed by t or $e(ea)$ with an attribute sequence ea captured by t. Given a thread t in P and an execution sequence x_p of the program P, the execution of t in this sequence is denoted as $x_p[t]$, i.e., $x^t = x_p[t]$.

The Property of Isolated Programs and Interface Verification

In this section we give a relationship between an isolated program execution and original thread execution. Then, we give a theorem that enables modular interface verification of a program. Recall that interface verification checks thread interface

correctness. Since there is only one thread at an isolated program, in the rest of the discussion, we will call an isolated program interface correct when the isolated thread is interface correct.

We first summarize the transition relation for an isolated program here. During an execution of an isolated program P' for an explicit thread or the main thread, the transition from one configuration to another is based on the relation $R^{P'} = R_{stub} \cup R_{si} \cup R_{sstub} \cup R_{stc} \cup R_{lop^t}$ where t is either the main thread or an explicit thread. During the execution of an isolated program for an RMI thread or the event thread, the transition from one configuration to another is based on the relation $R^{P'} = R_e \cup R_{stub} \cup R_{si} \cup R_{sstub} \cup R_{stc} \cup R_{lop^t}$ where t is either the event thread or an RMI thread. If there are asynchronous communication operations in the original program, R_{scomS} and R_{scomR} are added to the relations $R^{P'}$. An execution sequence of P' is $x_{p'} = x_0, x_1, \ldots, x_i, \ldots$ where $x_i = c \xrightarrow{label} c'$ with $(c, c') \in R^{P'}$ and $label$ is an operation or an event with any argument sequence. We denote the set of execution sequences of P', including the empty execution sequence, as $X_{P'}$.

Based on the single thread execution in the preceding discussion and the isolation techniques presented in this section, we give the following theorem.

Theorem 3.4.1 *Given a thread t of a program P and the isolated program P' with respect to t, $\{\Pi_1(x_p)(t) | x_p \in X_p\} \subseteq \{\Pi_1(x_{p'})(t) | x_{p'} \in X_{p'}\}$ where X_p is the set of executions of P and $X_{p'}$ is the set of executions of P'.*

During the isolation, the local operations are preserved and all other operations are abstracted. When we remove all of the controller variables from the $\mathcal{V}$ of the

original program and put the variable `cur`, then the transition relation for each abstracted operation is a superset of the transition relation for the original operation, and the transition relation for the local operations remains the same. Also, the transition relation for events in an isolated program subsumes the transition relation for events captured by a thread t i.e, $R_e^t \subseteq R_e$. Therefore, $R^t \subseteq R^{P'}$. Because of this subset relationship, the set of execution sequences of the thread t is subsumed by the set of execution sequences of the isolated program P'.

Let x^t be an execution of the thread t in the original program P where $x^t = x_p[t]$ and $x_p \in X_p$. The execution sequence x^t is an element of $X_{p'}$. Since $\Pi_1(x_p)(t)$ for $x_p \in X_p$ only concerns the executions of t, $\Pi_1(x_p)(t) = \Pi_1(x_p[t])(t)$. Therefore, the projection $\Pi_1(x_p)(t)$ is an element of $\{\Pi_1(x_{p'})(t) | x_{p'} \in X_{p'}\}$. ∎

Using this theorem we can claim that the interface verification result of t in isolated program P' holds on the t in the original program P and perform interface verification after we isolate each thread. We state this property as follows.

Theorem 3.4.2 *Given a program P and a set of isolated programs P'_t for each thread t of P, if every P'_t is interface correct then all of the threads in the original program P are interface correct, i.e., there is no interface violation in the original concurrent program P.*

We are going to prove this theorem by contradiction using Definition 3.3.1 and Theorem 3.4.1. Suppose the claim is not true, i.e., there is a thread t at P that is not interface correct and every isolated program is interface correct including the isolated program P'_t. Since t is not interface correct, there is an execution x_p and

a projection $\Pi_1(x_p)(t)$ which is not a legal sequence of the product of interfaces in P according to Definition 3.3.1. Let this product interface be I. Then, according to Theorem 3.4.1, this $\Pi_1(x_p)(t)$ is an element of $\{\Pi_1(x'_p)(t) \mid x'_p \in X'_p\}$ where X'_p is the set of executions of P'_t. Therefore, there is a execution projection of the isolated program P'_t which is not a legal sequence of I. This means that the isolated program P'_t is not interface correct which is a contradiction. ∎

Based on Theorem 3.4.2, we perform interface verification on each isolated program separately. This modularity improves the interface verification significantly by eliminating the possible thread interleavings.

A similar argument is valid for interface verification of composite web services.

Theorem 3.4.3 *Given a peer program P participating DP_A and the corresponding isolated program P' with peer interface I, $\{\Pi_4(x_{DP_A})(P)(I)|x_{DP_A} \in X_{DP_A}\} \subseteq \{\Pi_4(x_{p'})(P')(I) \mid x_{p'} \in X_{p'}\}$ where X_{DP_A} is the set of executions of DP_A and X'_p is the set of executions of P'.*

The proof is similar to the proof of Theorem 3.4.1. Similar to the proof of Theorem 3.4.1, the isolated peer program P' generates more conversations (i.e., has more behavior) than the original peer program DP_A since the relations R_{scomS} and R_{scomR} at P' subsume the relations R_{comS} and R_{comR} at DP_A. Since the projection Π_4 for an execution sequence is a conversation and each conversation of DP_A is a conversation of P', $\{\Pi_4(x_{DP_A})(P)(I)|x_{DP_A} \in X_{DP_A}\} \subseteq \{\Pi_4(x_{p'})(P')(I) \mid x_{p'} \in X_{p'}\}$ ∎

Theorem 3.4.4 *Given a program DP_A and a set of isolated peer programs P' for each participant program P, if every P' obeys its peer interface then there is no*

interface violation in the original distributed program DP_A.

The proof is similar to the proof of Theorem 3.4.2. If the claim is not correct, we can show a contradiction using Definition 3.3.5 and Theorem 3.4.3. ∎

3.5 How to Perform Interface Verification

We present how to find interface violations by using the model checker Java PathFinder (JPF) [99]. Interface verification is performed on isolated programs and uses controller interfaces encoded as finite state machines. This verification technique is used to check both thread interface correctness and obedience of peer implementations to their peer interfaces.

JPF is an explicit and finite state model checker for Java. It enables the verification of arbitrary pure Java implementations without any restrictions on data types. JPF supports property specifications via assertions that are embedded in the source code. It exhaustively traverses all possible execution paths for assertion violations. If JPF finds an assertion violation during verification, it produces a counter-example which is a program trace leading to that violation. In addition, JPF provides (currently) two nondeterminism utilities: `Verify.random` and `Verify.randomBool`. These utilities are used in stub and driver implementations discussed in Section 3.4 as well as in the finite state machine implementations. At verification time, JPF systematically analyze the program for every value created by these utilities.

The interface verification with JPF relies on the assertions in the finite state machine implementation `StateMachine` (see Figure 2.12 and 2.2). For concurrency

controllers, there are additional assertions in the stubs of shared data (`SharedStub` in Figure 2.2) that complements the controller interface specification. Recall that we can specify properties in JPF using assertions in the source code. We use this feature of JPF and perform interface verification with the assertions embedded in the `StateMachine` classes.

To perform interface verification with JPF, we substitute controller interface specifications for the controller classes. For peer controllers, we replace the asynchronous communication mechanism with `CommunicationInterface` in Figure 2.12. For the concurrency controllers, we replace the concurrency controller class with `ControllerStateMachine` in Figure 2.2 and shared data class with `SharedStub` class. Because of these substitutions, the action executions are directed to the methods of `StateMachine` that embed assertions. If there is an assertion violation, then there is an interface violation and JPF outputs the counter-example leading to that violation.

Now we describe this general methodology for peer controllers with more detail. The peer implementations are verified for one session since each session is independent and does not affect other sessions in the peer controller pattern. We replace `CommunicationController` with `CommunicationInterface` via source-to-source transformations (the classes are displayed in Figure 2.12). Since they both implement the same Java interface `Communicator`, there is no other modification necessary in the source code. In this transformed code, each send and receive operation is directed to the `sendTransition` and `receiveTransition` methods of `StateMachine` class that is used by the `CommunicationInterface`. The method

`sendTransition` invoked with a message instance to send. This method computes the set of next interface states using the `SendTransition` instances added by the `CommunicationInterface`. The method asserts that this set of next states is not empty. If the set is empty, then this transition is illegal and JPF reports the error with a counter-example. The method `receiveTransition` computes the set of possible receive transitions available in the current interface state. During this computation it uses the `ReceiveTransition` instances defined by the `CommunicationInterface`. This method then asserts that this set of possible receive transitions is not empty. A peer implementation conforms to its interface if JPF does not report any assertion violations.

For the concurrency controllers the details of this general methodology is as follows. We replace `Controller` classes with `ControllerStateMachine` classes implementing the same `ControllerInterface` Java interface and shared data classes with `SharedStub` classes implementing the same `SharedInterface` Java interface. With this transformation, the concurrency controller actions are directed to `transition` method of `StateMachine` and the shared data accesses are directed to the shared stub methods. Recall that the shared stub methods have assertions on the controller interface states. In addition, we isolate the concurrent threads with the thread isolation techniques presented in Section 3.4. With this process we can perform the interface verification on a single thread without considering the possible interleavings with other threads (see Theorem 3.4.1). JPF model checks this isolated program. When an action of concurrency controller is invoked, the `transition` method of `StateMachine` asserts that this action execution is valid at the current

interface state using the transitions defined by `ControllerStateMachine`. When a shared stub method is invoked, JPF checks if this access is allowed at the current interface state which is defined as an assertion within the `SharedStub` class. A concurrent thread implementation is interface correct if JPF does not report any assertion violations.

During the interface verification with JPF, we achieve improvements in the efficiency due to the following reasons. The peer interfaces are finite state machines and abstract away the asynchronous messaging details; therefore, using `CommunicationInterface` instead of `CommunicationController` reduce the state space of the system significantly. Another space reduction comes from considering only control attributes for message instances. The third factor in the reduction of the state space for web services is the verification of a peer implementation for one session. For concurrent programs, the efficiency is improved because of the state space reductions as well. The usage of controller interfaces instead of concurrency controllers reduce the state space dramatically since `ControllerStateMachine` classes do not contain the controller variables, locks and associated synchronization statements. Another factor is the usage of shared data stubs that do not contain actual data manipulation operations. Moreover, the thread isolation techniques enable us to perform interface verification on each thread separately. This isolation eliminates the need to consider all possible thread interleavings. This elimination leads to a significant improvement in the efficiency and scalability.

3.6 Composition of Interfaces

In the discussions above, we assumed that when a thread t uses more than one concurrency controller, say CC_1 and CC_2, t does not execute an action of CC_2 if it is not at the final state of the interface of CC_1. On the other hand, if we compose these two concurrency controllers, than we can interleave their actions. Moreover, we can also combine and execute their actions simultaneously. In this section, we discuss the composition of concurrency controllers.

Interfaces of different concurrency controllers can be composed to form more complex concurrency controllers [11, 12]. Recall that, a controller interface defines the rules about the order of controller action executions by each user thread. In an interface composition, actions of different concurrency controllers can be interleaved or they can be combined and executed simultaneously. We call the latter synchronous composition. We show that if the composed interface is a refinement of the original interface, then the ACTL properties of the original concurrency controller are preserved.

In this section, we will use the BB-RW example for illustration. This synchronization policy is realized as one concurrency controller in Section 2.1.1 and the discussions above are illustrated on this one concurrency controller realization. We could also specify this controller as a composition of BB and RW controllers.

The RW and the BB concurrency controllers can be composed in several different ways. Three of these compositions are given in Figure 3.2. In Figure 3.2(a), when a thread is writing, it could execute arbitrary number of `produce` and `consume`

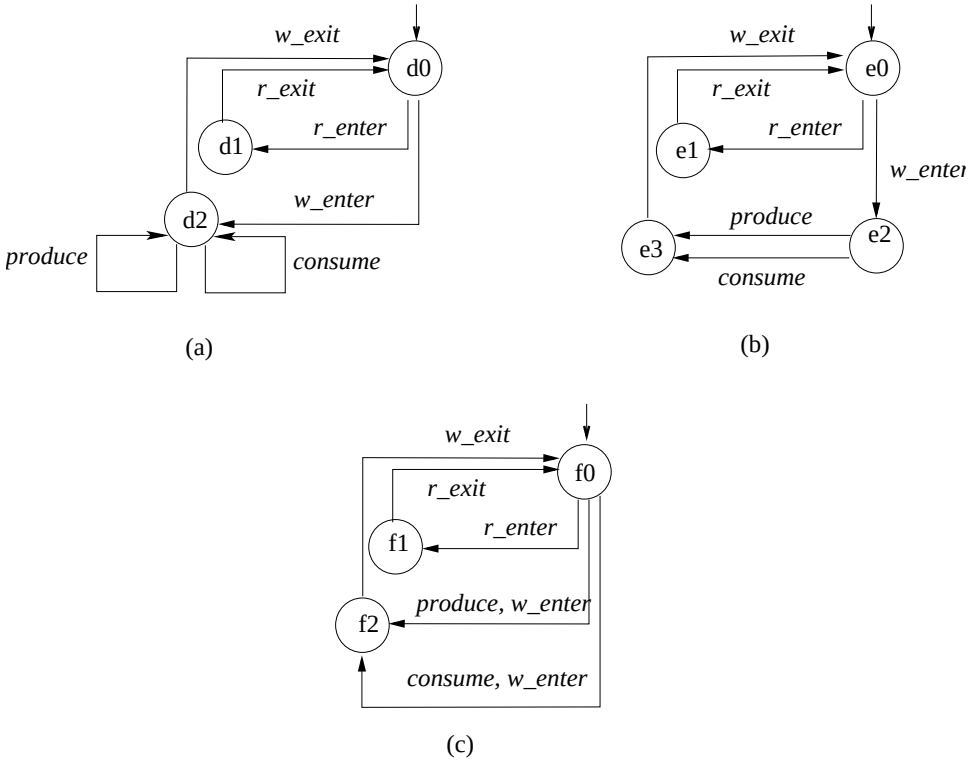


Figure 3.2: Composed interfaces

actions. In the interface given in Figure 3.2(b), however, when a thread is writing, it should execute either one `produce` or one `consume` action before it exits writing. There are two synchronously composed actions in Figure 3.2(c), one of them is the synchronous composition of `produce` and `w_enter` actions and the other one is the synchronous composition of `consume` and `w_enter` actions.

3.6.1 Composed Concurrency Controller Semantics

Let $CC_1, CC_2, \ldots, CC_k$ be k concurrency controller specifications with disjoint variables and actions, such that $CC_i = (\Gamma_i, IC_i, A_i, I_i)$ and $I_i = (Q_i, q_{0_i}, \{\}, A_i, \delta_i, F_i)$ for $1 \leq i \leq k$. Let $CC_c = (\Gamma_c, IC_c, A_c, I_c)$ be the composed concurrency controller where $\Gamma_c = \bigcup_{1 \leq i \leq k} \Gamma_i$, $IC_c = \bigwedge_{1 \leq i \leq k} IC_i$, $A_c = \bigcup_{1 \leq i \leq k} A_i$. The state space of a composed concurrency controller is defined similar to individual controllers defined in Section 3.2.1 by taking the Cartesian product of all the variable domains and the composed interface. The guards and the updates defining the actions of individual concurrency controllers are extended to the Cartesian product of the domains of all the variables in the composed component in a straightforward way: an update for an individual concurrency controller preserves the values of the variables of the other concurrency controllers. Note that, all parts of a composed concurrency controller other than its interface I_c is determined by the individual concurrency controllers that are composed.

Let the interface of the composed concurrency controller be $I_c = (Q_c, q_{0_c}, \{\}, \Sigma_c, \delta_c, F_c)$. Given $A_c = \bigcup_{1 \leq i \leq k} A_i$, the set of composed actions $CA \subseteq 2^{A_c}$ is defined as follows: For each composed action $ca = \{act_1, act_2, \ldots, act_r\} \in CA$, for each A_i, $|ca \cap A_i| \leq 1$ and $r \geq 1$, i.e., each composed action ca contains at most one action from each A_i. Then the input alphabet of the composed concurrency controller is $\Sigma_c = CA$. The transition relations are only send transitions with no guarding conditions and update functions, similar to the individual concurrency controllers. If a transition of a composed interface is labeled by a singleton set, then that transition corresponds to executing a single action from an individual component. On the other

hand, a transition which is labeled by more than one action corresponds to executing multiple actions from different concurrency controllers synchronously. Note that a composed action can have a mixed set of blocking and nonblocking actions. A thread executing a composed action has to wait until all the blocking actions become executable.

Let $T(CC_c)(n) = (IT, ST, RT)$ be the transition system of the composed concurrency controller $CC_c = (\Gamma_c, IC_c, A_c, I_c)$. The initial states IT and the set of states ST of the transition system $T(CC_c)(n)$ is based on Γ_c and IC_c similar to the semantics of individual concurrency controllers discussed in Section 3.2.1. To define the transition relation RT of a composed concurrency controller we need to define the semantics for the transitions of the form $(q, ca, q') \in \delta_c$ where $ca \in CA$. We do this by defining a set of composed update functions for each composed action. The composed update function corresponds to synchronous execution of all the individual actions in the corresponding composed action.

Given an action act_i, let $\text{IND}(act_i) = j$ if $act_i \in A_j$ (act_i is an action from the individual component CC_j). Given a state $s \in ST$, and a composed action $ca = \{act_1, act_2, \ldots, act_r\} \in CA$, $\text{COMP}(ca)(s)$ is defined as follows:

- If there exists a blocking action $act_i \in ca$ such that for all guarded commands $gc \in act_i.GC$, $gc.g(s(\Gamma))$ is FALSE, then $\text{COMP}(ca)(s) = \emptyset$.
- Otherwise, for all actions $act_i \in ca = \{act_1, act_2, \ldots, act_r\}$ choose a guarded command $gc_i \in act_i.GC$ such that $gc_i.g(s(\Gamma))$ is TRUE. If there is no such guarded command for an action act_i (which implies that act_i is non-blocking) let gc_i be gc_I, a guarded command with a TRUE guard and an identity update

function. The composed update function is

$$u^c(s(\Gamma)) = \bigwedge_{1 \leq i \leq r} gc_i.u(s(\Gamma_{\text{IND}(\mathit{act}_i)}))$$

The $\text{COMP}(ca)(s)$ is the set of all such composed update functions.

Given a transition (q, ca, q') we define the following:

$$RT^u_{(q,ca,q')} = \{(s, s') \mid (s, s' \in ST) \ \wedge \ (\exists u^c \in \text{COMP}(ca)(s), s'(\Gamma_c) = u^c(s(\Gamma_c)))$$
$$\wedge (\exists 1 \leq t \leq n, \ s(Q_c)(t) = q \ \wedge \ s'(Q_c)(t)q' \ \wedge \ s'(Q - t) = s(Q - t))\}$$

Finally, the overall transition relation RT of $T(CC_c)(n)$ is the following.

$$RT = \bigcup_{(q,ca,q') \in \delta_c} RT_{(q,ca,q')}$$

3.6.2 Refinement Relation

In this section, we define a refinement relation for interfaces. We will use this refinement relation to check whether a composition preserves properties verified on individual controllers. If a composed interface is a refinement of another interface then the ACTL properties verified on the original controller is preserved by the composed controller.

Let $CC_1, CC_2, \ldots, CC_k$, be k concurrency controller specifications, and let $CC_c = (\Gamma_c, \ IC_c, \ A_c, \ I_c)$ be a composition of these concurrency controllers and let I_c = $(Q_c, q_{0_c}, \ \{\}, \Sigma_c, \ \delta_c, F_c)$. be the composition interface. We use $I_c \preceq I_i$ to denote that the composed interface I_c is a refinement of the interface I_i.

Definition 3.6.1 *$I_c \preceq I_i$ if and only if there exists a mapping $H : Q_c \rightarrow Q_i$ such that for all $q_c \in Q_c$ and for all $q \in Q_i, H(q_c) = q$ implies that, $q_c = q_{0_c} \Rightarrow q \in q_{0_i}$ and for each $(q_c, ca, q'_c) \in \delta_c$,*

$$
\begin{aligned}
&(a \in ca \ \wedge \ a \in A_i \Rightarrow \exists (q, a, q') \in \delta_i, H(q'_c) = q') \\
&\wedge (ca \cap A_i = \emptyset \Rightarrow \exists (q, \epsilon, q') \in \delta_i, H(q'_c) = q') \\
&\wedge (ca \cap A_i \neq \emptyset \ \wedge \ (\exists j \neq i, b \in ca \cap A_j \ \wedge \ blocking(b)) \Rightarrow \exists (q, \epsilon, q) \in \delta_i)
\end{aligned}
$$

Let CC_i be one of the controllers in the composition of the composed controller CC_c. Let $T(CC_c)(n) = (IT_c, ST_c, RT_c)$ be the transition system for CC_c and $T(CC_i)(n) = (IT_i, ST_i, RT_i)$ be the transition system for CC_i. Given a mapping $H : Q_c \rightarrow Q_i$ between the interface states of the CC_c and CC_i we define a projection function $\Psi : ST_c \rightarrow ST_i$ such that, given $s_c \in ST_c$, $\Psi(s_c)(\Gamma_i) = s_c(\Gamma_i)$ and for all $1 \leq t \leq n$, $\Pi(s_c)(Q_i)(t) = H(s_c(Q)(t))$. Observe that, for any atomic property $p \in AP_i$, $s_c \models p$ if and only if $\Psi_i(s_c) \models p$. We can generalize the projection function to paths such that given a path $exe^c = s^c_0, s^c_1, \ldots$ in $T(CC_c)(n)$, $\Psi(exe^c) = \Psi(s^c_0), \Psi(s^c_1), \ldots$.

Lemma 3.6.2 *If $I_c \preceq I_i$, then for all paths exe of CC_c, the projection $\Psi(exe)$ is a path in CC_i where projection Ψ is based on the mapping function H from Definition 3.6.1 that shows that $I_c \preceq I_i$.*

Based on the above lemma we can prove the following theorem:

Theorem 3.6.3 *Given a concurrency controller $CC_c = (\Gamma_c, IC_c, A_c, I_c)$ which is a composition of concurrency controllers $CC_1, CC_2, \ldots, CC_k$, CC_c preserves all ACTL properties of CC_i, $1 \leq i \leq k$, if $I_c \preceq I_i$.*

Let Ψ denote the refinement projection based on the mapping function H from Definition 3.6.1 that shows that $I_c \preceq I_i$. In order to prove that the ACTL properties of CC_i are preserved in CC_c it is sufficient to show that for all $s \in ST_c$, $\Psi(s) \models \mathrm{A}f \Rightarrow s \models \mathrm{A}f$ assuming that f is of the form Fp, Gp, pUq, and $\Psi(s) \models p \Rightarrow s \models p$ and $\Psi(s) \models q \Rightarrow s \models q$. Note that $\Psi(s) \models p \Rightarrow s \models p$ when p is an atomic property of the concurrency controller CC_i. If we can prove the claim above, we can show by structural induction that Theorem 3.6.3 holds.

We are going to prove the above claim using proof by contradiction. Suppose the claim is not true, i.e. $\Psi(s) \models \mathrm{A}f \wedge s \not\models \mathrm{A}f$. Note that, $s \not\models \mathrm{A}f$ implies that $s \models \mathrm{E}\neg f$ which means that there exists a path exe in CC_c starting at s which satisfies $\neg f$. Then, according to the Lemma 3.6.2, there exists a path $\Psi(exe)$ in CC_i starting at $\Psi(s)$ which satisfies E$\neg f$ which is a contradiction. ∎

Using Theorem 3.6.3, we are able to verify the ACTL properties of the individual concurrency controllers modularly before composing them. If we can show that the refinement relation holds, then the verified properties are preserved in the composed system. However, above theorem cannot be used for properties that refer to variables of different individual concurrency controllers. Such a property can be verified on the transition system of the composed concurrency controller CC_i after the composition.

3.7 Related Work

De Alfaro et al. [31] introduce interface automata to formalize temporal aspects of software component interfaces. Later, this formalization is used for specifying interfaces as a set of constraints and algorithms for interface compatibility checking is presented by Chakrabarti et al. [27, 18]. We use finite state machines to specify interfaces and our approach to interface checking can be seen as a special case of the interface compatibility checking. However, unlike Chakrabarti et al., we do not require each method to be annotated with interface constraints and also our goal is to verify both the controller behavior and conformance to interface specifications.

Whaley et al. [100] also model interfaces as finite state machines. The authors extract such interfaces from programs using both static and dynamic analysis. Their approach is, however, a reverse engineering approach where interfaces are automatically extracted from existing code. We use interfaces both for specification and automated verification.

Rajamani et al. [86] address the conformance of an implementation model to a specification for asynchronous message passing programs. Unlike the interface verification in our framework, their conformance check requires a model extraction from the implementation. Moreover, their approach does not separate the interface and the behavior verification steps.

There has been some work on thread modular verification and automated environment generation. Flanagan et al. [44] presents a thread-modular reasoning and verifies each thread separately with respect to safety properties. The effects of other

threads are modeled as environment assumptions whereas we use stubs and drivers to reflect these effects. Besides, we check the thread behavior against the interface rules and leave the assurance of the safety properties to behavior verification.

Godefroid et al. [50] converts an open reactive program into to a closed program by inserting nondeterminism into the code and eliminating procedure arguments. Unlike this work, we have restrictions on the environment interactions caused by controllers via interfaces. Stoller [94] transforms distributed programs communicating with RMI into one program for model checking. Unlike this centralization approach, we apply thread modular model checking, decouple the remote processes, and reduce the state space. The techniques presented by Tkachuk et al. [95, 96] generate environments for software components by using side effect and points-to analyses. Although the techniques we discuss for thread isolation are similar to these, we base our techniques on the controller interfaces and the design for verification approach.

The graphical user model introduced by Dwyer et al. [38] is similar to our generic GUI driver. Unlike our GUI driver, their model creates all types of user events after choosing a GUI object. The actual event thread in Java, on the other hand, dispatches only one user event at a time. The other difference is that our driver is used for interface verification whereas their model is used for analyzing user interaction orderings.

Data independence is first formalized by Wolper [101]. A data independent program does not use a certain set of values to make control decisions. According to Wolper, when the program uses such values only to input, copy, and output, we can

replace the domain of this data with a smaller and finite set. Lazic [67] extends the data independence to allow values of data independent variables to appear in equality testing. These studies focus on identifying the data types of which the general behavior of the program is independent. We are interested in determining the elements of argument sequence of which the synchronization behavior of the program is independent.

Chapter 4

Design for Verification of Concurrent Programming

This chapter presents the design for verification (DFV) approach for concurrent programming based on the concurrency controller pattern. The goal of this approach is to eliminate synchronization errors in a concurrent program. In this chapter, we present an assume-guarantee style verification technique based on the concurrency controller pattern given in Section 2.1 in Chapter 2. We show that model checking techniques can be effective in finding synchronization errors in safety critical software when they are combined with the DFV approach.

Our assume-guarantee verification approach presented in this chapter exploits the modularity of the concurrency controller design pattern, i.e., decoupling of the concurrency controller behavior from the threads that use the controller. The behavior of a concurrency controller can be verified with respect to arbitrary number of threads and with unbounded variables and parameterized constants using infinite state model checking techniques. The threads that use the controller classes can be

checked for interface violations using finite state model checking techniques which allow verification of arbitrary thread implementations without any restrictions.

In this chapter, we also demonstrate the application of the DFV with concurrency controllers to two real-life concurrent software systems. The first one is a Concurrent Editor implementation with remote procedure calls and complex synchronization constraints that uses a client-server architecture. The Concurrent Editor allows multiple users to edit a document concurrently as long as they are editing different paragraphs. Concurrent Editor maintains a consistent view of the shared document among the client nodes and the server. The second one is a safety critical air traffic control software called Tactical Separation Assisted Flight Environment (TSAFE). On this safety critical software we have conducted an experimental study investigating the effectiveness of the presented DFV approach. We have reengineered TSAFE, created 40 faulty versions of TSAFE using fault seeding, and used our verification approach for finding the seeded faults.

The organization of this chapter is as follows. Section 4.1 introduces the Concurrent Editor. Section 4.2 discusses TSAFE and the reengineered version in detail. Section 4.3 explains the assume-guarantee style modular verification based on the concurrency controller pattern. This section also presents the verification results for both software systems and discusses the experimental study performed on TSAFE.

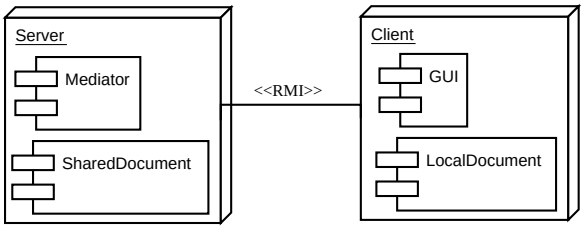


Figure 4.1: Concurrent Editor architecture

4.1 Concurrent Editor

Concurrent Editor is a distributed system that consists of a server node and a number of client nodes (one client node per user). In other words, the Concurrent Editor is a $DP = (P_1, P_2, \ldots, P_k)$ where P_1 is the program running on the server node and $P_2, \ldots, P_k$ are the programs running on client nodes. Each node in this structure has its own copy of the shared document (Figure 4.1).

Concurrent Editor allows multiple users to edit a document concurrently as long as they are editing different paragraphs and maintains a consistent view of the shared document among the client nodes and the server. Users get write access to paragraphs of the document by clicking on the **WriteLock** button in the graphical user interface (GUI). Figure 4.2 shows a screen shot. When **WriteLock** button is clicked, it generates a request for write access to the paragraph the cursor is on. When the request is granted, the color of the paragraph is changed to indicate that it is editable. A user can edit a paragraph only if she has the write access to that paragraph. Multiple users are able to edit different paragraphs of the document concurrently, i.e., each user is able to see the changes made by the other users as they occur. If a user

wants to make sure that a paragraph does not change while reading it, she clicks the **ReadLock** button in the GUI. When a user has read access to a paragraph, other users can also have read access to that paragraph, but no other user can have write access to that paragraph. When a user has write access to a paragraph, no other user can have read or write access to that paragraph. All users have a copy of (and can see) the whole document, including the paragraphs they do not have read or write access. The GUI also provides **ReadUnlock** and **WriteUnlock** buttons to release the read and write accesses, respectively. Finally, the document is saved only when a consensus is reached among all client nodes.

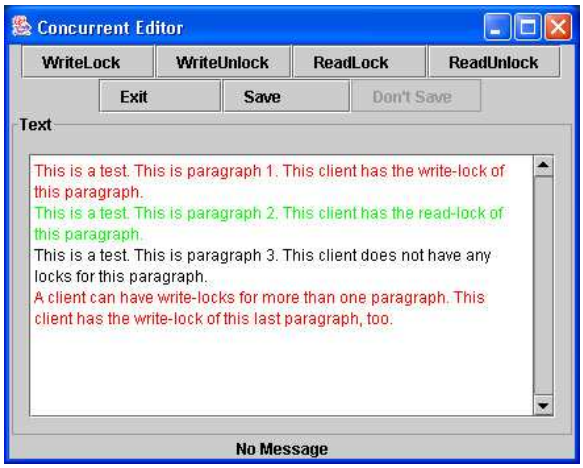


Figure 4.2: Concurrent Editor screen shot

Designing and Implementing the Concurrent Editor

When the above specification is given to a Java programmer, she has to consider both concurrent and remote accesses to the shared document. One can handle the

remote access using the Java remote method invocation (RMI) and a collaborative infrastructure where all client nodes register to a mediator through which the clients communicate [81] (see Figure 4.1).

A number of synchronization issues need to be considered to handle the concurrent access to the shared document. Let us first focus on coordinating concurrent access to a single paragraph in the shared document. A naive solution is to declare all methods of the paragraph as `synchronized`, which is obviously not efficient. If mutual exclusion is enforced at every method, a client node requesting a read access will be blocked by another client node requesting the same access. The programmer needs to use a reader-writer lock (RW) to achieve a more efficient synchronization. A common methodology is encapsulating the synchronization policy within the shared data implementation, e.g., implementing the `write` method of the paragraph so that it acquires and releases the write lock implicitly. However, this methodology contradicts with the requirements of the Concurrent Editor, since it forces the write lock to be acquired at every write request. Therefore, the programmer needs to separate the lock acquisition from the actual write operation. A welcome side effect of this separation is that the programmer can change the paragraph implementation without affecting the synchronization policy. Similarly, one synchronization policy could be replaced with another without changing the paragraph implementation.

Here we describe another synchronization constraint. While protecting the shared document, we cannot afford to suspend a client. Otherwise, the whole application will stall due to the collaborative infrastructure. To prevent the stalling of a client node, we need a separate worker thread to acquire the locks from the server node.

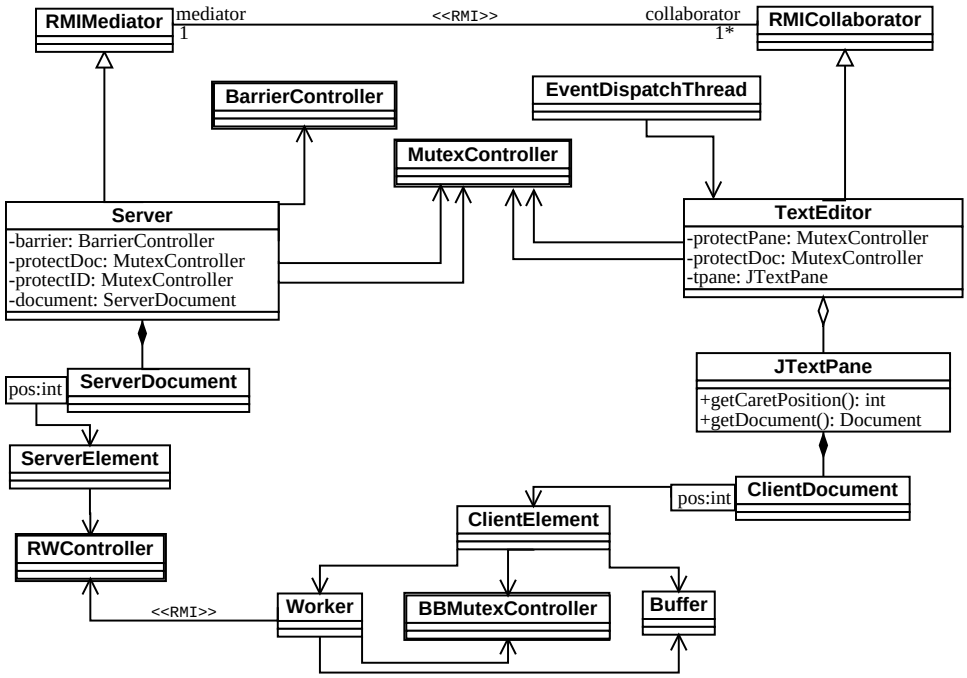


Figure 4.3: Concurrent Editor class diagram

This way, whenever the server node forces the requesting thread to wait it does not suspend the client node. The use of a worker thread generates the need for a buffer for passing the requests between the client node and the worker threads, and the buffer needs to be protected by a synchronization policy. It is clear that the accesses to this buffer should be mutually exclusive. In this application, however, a mutual-exclusion (mutex) lock is not enough. We need a conditional wait for worker threads when the buffer is empty. We also need to put a bound to the buffer size to provide a reasonable response time to the user. Therefore, we protect this buffer by using a bounded buffer synchronized with a mutex lock (BB-MUTEX). In this synchronization policy, if a thread wants to take an item from the buffer it would wait while the

buffer is empty. This policy also restricts the number of threads accessing the buffer, e.g., at any given time only one producer or consumer thread can access the buffer.

We have implemented the Concurrent Editor based on the concurrency controller pattern while considering the above constraints and issues. Figure 4.3 shows the class diagram of our Concurrent Editor implementation using the concurrency controller pattern. The server node has a document of type `ServerDocument`. Mutually exclusive access to server document is ensured by a mutex controller (MUTEX). The server document consists of a number of paragraph elements. Each paragraph is associated with a unique reader-writer controller (RW) coordinating the read and write accesses to that paragraph. This node also has a barrier controller (BARRIER) which is used whenever a save request is issued by one of the users. A document is saved to disk only if a consensus is reached among all the users of the document.

A client node has a GUI with an editable text area and seven buttons (see Figure 4.2). The text area is of type `JTextPane` containing a document of type `ClientDocument`. Both the text pane and the document are protected with mutex controllers (MUTEX). The text of the client document is a copy of the server document. There is an event thread (`EventDispatchThread`) running on the client side. The event thread is automatically created by the Java Virtual Machine (JVM) to capture the events related to the GUI [82]. In addition to the event thread, each paragraph element in the client document is associated with a unique worker thread created by the event thread. Each worker thread handles the communication with the RW controller of the corresponding paragraph in the server node. The communication between the event thread and the worker thread for a paragraph is via a communica-

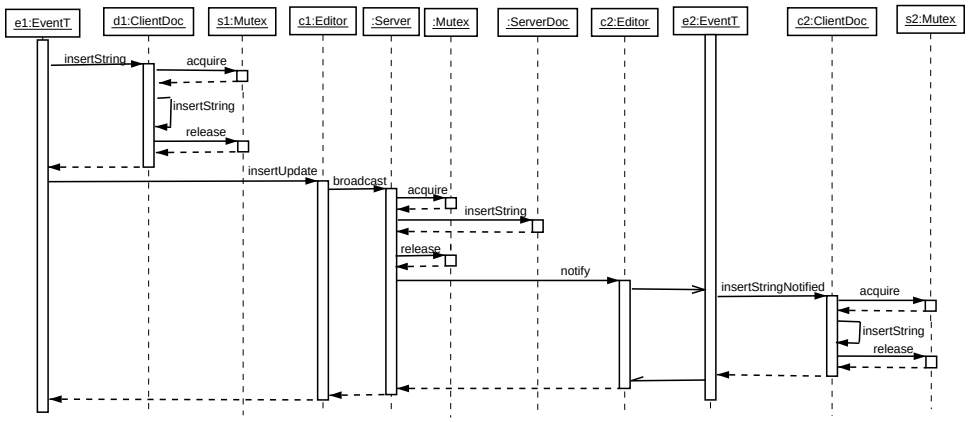


Figure 4.4: Sequence diagram for inserting text

tion buffer associated with that paragraph. The access to this buffer is controlled by a bounded-buffer mutex controller (BB-MUTEX).

Figure 4.4 shows a sequence diagram that represents the behavior of the Concurrent Editor during an insertion of a character into the text. For simplicity, the sequence diagram shows two client nodes and the server. We assume that the client node that does the character insertion has the write access to the paragraph. The event thread on client node 1 inserts the character typed by the user into the client document. Then, the event thread of client 1 makes a remote call to the server's `broadcast` method. On the server side, this invocation results in a change in the server document and an RMI call to other client's `notify` method. On the second client node's side, the character is inserted into the client document after acquiring the mutex lock.

The Concurrent Editor implementation consists of 2800 lines of Java code with 17 class files, 5 controller classes, and 7 Java interfaces. We used concurrency con-

trollers to coordinate the interaction among multiple threads. Hence, we implemented the Concurrent Editor without writing any Java synchronization operations. In implementing the Concurrent Editor we used the following controllers: mutex controller (MUTEX), reader-writer controller (RW), barrier controller (BARRIER), and bounded-buffer with mutex lock controller (BB-MUTEX). We have discussed the usage of these controllers above. The interfaces of these controllers are given in Figure 4.5. For example, the interface of the RW controller has three states: `IDLE`, `READ`, and `WRITE` with `IDLE` being the initial state.

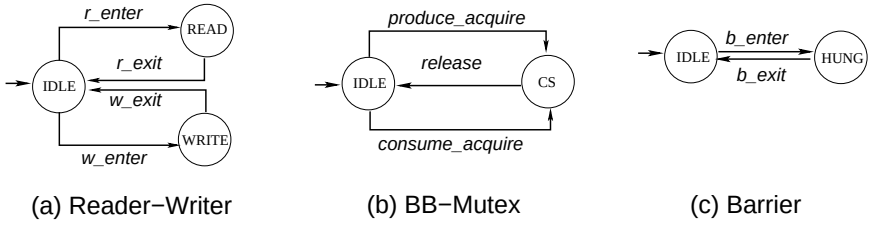


Figure 4.5: Controller interfaces used in Concurrent Editor

4.2 TSAFE

The Automated Airspace Concept that is being developed by NASA researchers automates the decision-making in air traffic control by giving the responsibility of determining conflict free trajectories for aircraft to a software system [39, 40]. Establishing dependability of such complex systems is extremely difficult, yet it is essential for automation in this domain. Earlier efforts in automating the air traffic control system have resulted in costly failures due to the inability of the contrac-

tors in making the software components highly dependable [41]. To avoid a similar fate, the designers of the Automated Airspace Concept at NASA use a *failsafe short term conflict detection component* in their system, which is responsible for detecting conflicts in flight paths of the aircraft within one minute from the current time. If a short term conflict is detected, this component takes over the trajectory synthesis function to direct the aircraft to a safe separation. Since the goal of this component is to provide failsafe conflict detection and resolution capability, it has to be highly dependable, even more than the rest of the system [39, 40].

The Tactical Separation Assisted Flight Environment (TSAFE) software is a partial implementation of this component. Based on the design proposed by NASA researchers a version of TSAFE was implemented at MIT [36]. Later on, as a part of the NASA's High Dependability Computing Project, TSAFE software was integrated into an experimental environment at the Fraunhofer Center for Experimental Software Engineering, Maryland [69]. The TSAFE experimental environment contains software artifacts including requirements specifications, design documentations, source code (Java), as well as faults that can be seeded into various artifacts for several versions of TSAFE.

We used a distributed client-server version called TSAFE III that performs the following functions: 1) Display aircraft position, i.e., indicate where the aircraft is located at a certain time, 2) Display aircraft planned route, i.e., indicate the route that the aircraft intends to follow according to the flight plan, 3) Display aircraft future projected route trajectory, i.e., display the probable trajectory that the aircraft will follow, 4) Indicate conformance problems, i.e., indicate whether a flight is conform-

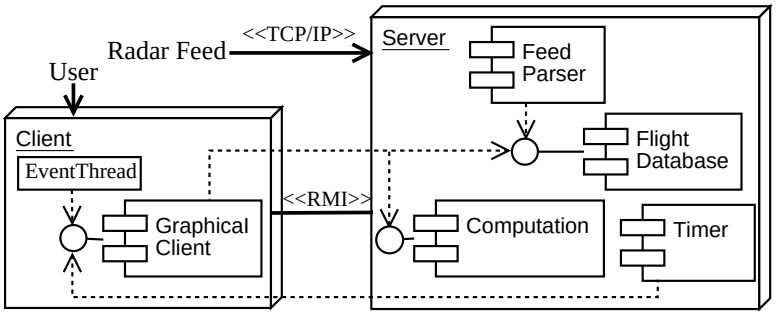


Figure 4.6: TSAFE architecture

ing to the planned route or blundering.

The TSAFE III implementation consists of 21,057 lines of Java source code in 87 classes. Figure 4.6 shows the architecture of TSAFE III. The server stores the trajectories of the flights in a flight database. The feed parser thread in the server receives updates of the locations of the flights periodically from the radar feed through a network connection and updates the trajectory database. A computation component in the server implements the trajectory synthesis and conformance monitoring functions. The client implements the display functionality in a GUI. Multiple clients can connect to the server at the same time via RMI. A timer thread at the server periodically prompts the clients to access the flight database to obtain the current data.

Reengineering TSAFE

We reengineered the TSAFE software as follows: 1) We identified all the synchronization statements (`synchronized`, `wait`, `notify`, `notifyAll`) in the TSAFE code and we also identified the shared data these statements are used to protect, 2)

We developed the concurrency controllers implementing the synchronization policy required for accessing these shared data, 3) We replaced all the synchronization statements in the TSAFE code with calls to the appropriate concurrency controller classes. All the synchronization statements in the reengineered TSAFE code are in the helper classes provided by the concurrency controller pattern.

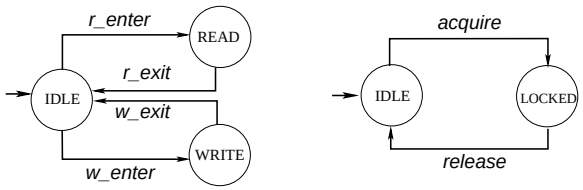
Here we discuss one of the synchronization requirements and the concurrency controller solution for that requirement. In TSAFE, the flight database is accessed by multiple threads which may cause failures in its functionality if the threads are not properly synchronized. For example, while the thread running the feed parser is updating the trajectory database, a thread serving an RMI call from a client may be reading it. If such an interaction occurs, an aircraft's location may be displayed incorrectly on the client GUI. Since the client GUI provides the interaction between the human air traffic controllers and the TSAFE system, displaying incorrect information on it could have disastrous effects. To prevent such synchronization problems in Java programs, Java programmers declare the methods of such classes to be synchronized. However, this is not an efficient solution for this case. If the methods of the database are synchronized, then at any given time at most one client thread can access the database. Since client threads never update the database, this synchronization is unnecessary and may slow down the GUI displays. A more appropriate synchronization policy for such cases is to use a reader-writer lock. Using a reader-writer lock multiple readers can access a shared resource at the same time, but a writer can access the shared resource only alone. A programmer can implement this solution and protect the flight database with an RW controller. The programmer also

needs to make sure that the appropriate methods of the reader-writer lock class are called before accessing the database.

The flight database implementation in TSAFE is called `RuntimeDatabase`. The methods of the `RuntimeDatabase` class were synchronized in the original version. Two of these methods are: `insertFlight`, which updates the database by inserting a flight, and `selectFlight`, which is used to read the information about a flight. In the reengineered code the methods of the `RuntimeDatabase` are not synchronized. During the reengineering process, we specified the constraints on accessing the flight database based on the interface states of the RW controller with a `RuntimeDatabase_Stub`. (This class corresponds to the `SharedStub` in the pattern diagram given in Figure 2.2). With the assertions in the methods, this stub class specifies that, for example, a thread has to call `w_enter` before calling `insertFlight` and it has to call `w_enter` or `r_enter` before calling the method `selectFlight`.

In the reengineered TSAFE code there are two concurrency controller classes. One of them is the RW controller described above. The other one is a MUTEX controller implementing a mutex lock with `acquire` and `release` actions. In the reengineered TSAFE code, there are 2 instances of the RW controller and 3 instances of the MUTEX controller protecting 6 shared data instances.

The interfaces of the two concurrency controllers we used while reengineering TSAFE are shown in Figure 4.7. The interface of the MUTEX controller shown in Figure 4.7(b) has two states: `IDLE` and `CS` with `IDLE` being the initial state. The RW controller is the same concurrency controller we have used for the Concurrent



(a) Reader−Writer (b) Mutex

Figure 4.7: Controller interfaces used in reengineered TSAFE

Editor. The interface of this controller shown in Figure 4.7(a) has three states with `IDLE` being the initial state.

4.3 Verification of Concurrency Controllers

In this section, we present our modular and assume-guarantee style verification approach based on the concurrency controller pattern. The verification consists of two steps: 1) *Behavior verification:* Verification of the properties of the concurrency controller classes assuming that the user threads obey to the controller interfaces; 2) *Interface verification:* Verification of the threads that use the concurrency controllers to make sure that they access the methods of the controllers and the shared data in the order specified by the controller interfaces and the data stubs.

4.3.1 Behavior Verification

We verify the behavior of a concurrency controller using the Action Language Verifier [23]. We automatically translate the concurrency controllers written based on the concurrency controller pattern into Action Language. The behavior veri-

fication is based on the concurrency controller semantics and the projections discussed in Chapter 3, i.e, the transition system $T(CC)(n)$, where n user threads and $CC = (\Gamma, IC, A, I)$ is the concurrency controller, and the projection function Π_2. (See Section 3.2 and Section 3.3.2 for more detail.)

Action Language Verifier (ALV) is an infinite state symbolic model checker and can verify specifications with unbounded integer variables. ALV takes a specification in Action Language and a set of Computation Tree Logic (CTL) formulas as input. This model checker uses conservative approximation techniques such as widening and truncated fixpoint computations to conservatively verify or falsify infinite state specifications with respect to the given CTL formulas [23].

We use the BB-RW controller discussed in Chapter 2 to demonstrate the verification approach. An implementation of this concurrency controller (`BBRWController`) with five actions and four controller variables is given in Figure 2.3. The controller interface for BB-RW is given in Figure 2.6(a). Recall that the behavior of a concurrency controller is specified using the instances of Action class in a guarded command style, and the required execution order of these actions are specified with its controller interface. Using this ordering and the definition of actions, each controller action is automatically translated to the atomic actions of the Action Language. The automatically generated Action Language specification for the BB-RW is given in Figure 4.8.

An atomic action in the Action Language defines one execution step using current-state values for the variables (denoted as unprimed variables) and next-state values (denoted as primed variables). Our tool translates the guard expression to a formula

```
module main()
 integer nR; boolean busy; integer count;
 parameterized integer size;
 module BBRW()
  enumerated pc {IDLE,READ,WRITE};
  initial: nR=0 and busy=true and pc=IDLE;
  r_enter: pc=IDLE and !busy and nR'=nR+1 and pc'=READ;
  r_exit_0: pc=READ and true and nR'=nR-1 and pc'=IDLE;
  r_exit_1: pc=READ and !(true) and nR'=nR-1 and pc'=IDLE;
  w_enter_produce: pc=IDLE and nR=0 and !busy and count<size
        and busy'=true and count'=count+1 and pc'=WRITE;
  w_enter_consume: pc=IDLE and nR=0 and !busy and count>0
        and busy'=true and count'=count-1 and pc'=WRITE;
  w_exit_0: pc=WRITE and true and busy'=false and pc'=IDLE;
  w_exit_1: pc=WRITE and !(true) and busy'=false and pc'=IDLE;
  BBRW: r_enter | r_exit_0 | r_exit_1 | w_enter_produce
      | w_enter_consume | w_exit_0 | w_exit_1;
 endmodule
 main: BBRW()| BBRW()| BBRW() | BBRW();
endmodule
```

Figure 4.8: Automatically generated Action Language specification for BB-RW

on current-state values, and the update expression to a formula on current-state and next-state values. As an example, consider the blocking action `w_enter_produce` in the `BBRWController` class. In the constructor, this action is defined with one guarded command. The guard expression of this guarded command is `(nR==0 && !busy && count>size)`, and the update expression is `busy = true; count = count + 1`. In the interface of the controller the transition associated with `w_enter_produce` changes the thread's state from `IDLE` to `WRITE`. Using this information, the translator constructs the atomic action `w_enter_produce` shown in the automatically generated Action Language specification in Figure 4.8. A nonblocking action in the concurrency controller class is translated into two atomic actions:

one represents the execution when the guard is true, and the other represents the execution when the guard is false. For example, the nonblocking `w_exit` action in the `BBRWController` class is translated into `w_exit_0` and `w_exit_1` in the Action Language specification shown in Figure 4.8.

A controller specification in Action Language consists of a main module and a submodule. Each instantiation of the submodule corresponds to a thread. The variables of the main module correspond to the controller variables of the concurrency controller. In the above example, the `main` module has a submodule called `BBRW`. Submodule `BBRW` models the user threads and corresponds to a process type (a thread class in Java). Submodule `BBRW` has one local enumerated variable. This enumerated variable (pc) keeps track of the thread state which is represented by a state of the controller interface. This is the only information we need to know about a thread to verify the controller implementation. A thread can be in one of the interface states. Each instantiation of a module will create different instantiations of its local variables.

Since Action Language Verifier can handle infinite state systems we are able to verify concurrency controllers with unbounded integer variables (such as `nR`) or parameterized constants (such as `size`). When a specification has a parameterized constant it is verified for all possible values of, for example, `size`. Currently, the variable types supported by the Action Language are integer, boolean, enumerated and heap. Therefore, we have to restrict the controller variables to these types to verify them with ALV (we use static integers as enumerated variables in the controller implementations). On the other hand, since controller variables only need to

store the state information required for synchronization, these basic types have been sufficient for modeling concurrency controllers we have encountered so far.

Parameterized Verification

We use an automated abstraction technique, called counting abstraction [34], to verify the behavior of a concurrency controller with respect to arbitrary number of threads. Implementation of counting abstraction for Action Language specifications is presented by Yavuz-Kahveci et al. [105]. The basic idea is to define an abstract transition system in which the local states of the threads (corresponding to the states of the interface) are abstracted away, but the number of threads in each interface state is counted by introducing a new integer variable for each interface state.

Given a concurrency controller $CC = (\Gamma, IC, A, I)$ with an interface $I = (Q, q_0, V, \Sigma, \delta, F)$, the parameterized transition system for that concurrency controller $T(CC)(p) = (IT_p,\ ST_p,\ RT_p)$, where p is a parameterized constant, is defined as follows: The set of states of the parameterized system is defined as $ST_p = \prod_{\gamma \in \Gamma} \text{DOM}(\gamma) \times \prod_{cv \in CV} \text{DOM}(cv) \wedge\ (\sum_{cv \in CV} cv = p)$, where CV is the set of integer variables introduced by the counting abstraction. Each $cv \in CV$ corresponds to an interface state $q \in Q$ and represents the number of threads in the interface state q. The initial states and the transition relation of the parameterized system can be defined using linear arithmetic constraints on these new variables [105]. Below is the parameterized Action Language specification generated for BB-RW.

```
module main()
 integer nR; boolean busy; integer count;
 parameterized integer size;
```

```
 parameterized integer numInstance;
 module BBRW()
  integer IDLE,READ,WRITE;
  initial: nR=0 and busy=true;
  initial: IDLE=numInstance and READ=0 and WRITE=0;
  restrict:numInstance>0;
  r_enter: IDLE>0 and !busy and nR'=nR+1
        and READ'=READ+1 and IDLE'=IDLE-1;
  r_exit_0: READ>0 and true and nR'=nR-1
        and IDLE'=IDLE+1 and READ'=READ-1;
  r_exit_1: READ>0 and !(true) and nR'=nR-1
        and IDLE'=IDLE+1 and READ'=READ-1;
  w_enter_produce: IDLE>0 and nR=0 and !busy and count<size
        and busy'=true and count'=count+1
        and WRITE'=WRITE+1 and IDLE'=IDLE-1;
  w_enter_consume: IDLE>0 and nR=0 and !busy and count>0
        and busy'=true and count'=count+1
        and WRITE'=WRITE+1 and IDLE'=IDLE-1;
  w_exit_0: WRITE>0 and true and busy'=false
        and IDLE'=IDLE+1 and WRITE'=WRITE-1;
  w_exit_1: WRITE>0 and !(true) and busy'=false
        and IDLE'=IDLE+1 and WRITE'=WRITE-1;
  BBRW: r_enter | r_exit_0 | r_exit_1 | w_enter_produce
      | w_enter_consume | w_exit_0 | w_exit_1;
 endmodule
 main: BBRW();
endmodule
```

In this parameterized specification, for example, the integer variable `IDLE` denotes the number of threads in the interface state `IDLE`. A parameterized integer constant, `numInstance`, denotes the number of threads. This parameterized constant is restricted to be positive. When the specification is verified with ALV the results hold for any valuation of this parameterized constant (i.e. the results are valid for any number of threads).

In the rest of this chapter, we call the Action Language specification generated by using the counting abstraction technique as an *abstract* specification, and the speci-

Table 4.1: Properties of the RW controller

RP1	$AG(busy \Rightarrow nR = 0)$
RP2	$AG(busy \Rightarrow AF(\neg busy))$
RP3	$AG(\neg busy \wedge nR = 0 \Rightarrow AF(busy \vee nR > 0))$
RP4	$\forall x\ AG(nR = x \wedge nR > 0 \Rightarrow AF(nR \neq x))$
RP5	$AG(pc = WRITE \Rightarrow AF(pc = IDLE))$
RP6	$AG(\neg(pc1 = READ \wedge pc2 = WRITE))$
RP7	$AG(\neg(pc1 = WRITE \wedge pc2 = WRITE))$
RP8	$AG(pc1 = READ \Rightarrow nR > 0)$
RP9	$AG(pc1 = WRITE \Rightarrow busy)$
RP10	$AG(WRITE > 0 \Rightarrow AF(WRITE = 0))$
RP11	$AG(\neg(READ > 0 \wedge WRITE > 0))$
RP12	$AG(\neg(WRITE > 1))$
RP13	$AG(WRITE = 1 \Leftrightarrow busy)$
RP14	$\forall x\ AG(READ = x \wedge READ > 0 \Rightarrow AF(READ \neq x))$
RP15	$AG(READ = nR)$

fication generated without using the counting abstraction as a *concrete* specification.

Concurrency Controller Properties

In order to verify the concurrency controllers with ALV we need a list of properties to specify the correct behavior of the controllers. We allow the CTL properties for the controllers to be either inserted directly to the generated Action Language specification or written as annotations in the controller classes (which are then automatically inserted into the Action Language translation).

The properties for the RW controller are shown in Table 4.1. The properties RP1–4 only refer to the controller variables. For example, the global property RP1 states that whenever `busy` is true `nR` must be zero. The remaining properties refer to both the controller variables and to the states of the threads. Note that the representation of the thread state is different in the concrete and the abstract Action Language specifications. The properties RP5–9 are for concrete specifications and refer to concrete thread states. For example, the property RP5 states that whenever a thread is in the `WRITE` state it will eventually reach the `IDLE` state. The properties RP10–15 are for the parameterized instances and refer to the integer variables which represent the number of threads in a particular state. For example property RP15 states that at any time the number of threads that are in the reading state is the same as the value of the variable `nR`. Note that, two of the properties shown in Table 4.1 contain universally quantified integer variables. We are able to check such properties using ALV by declaring the universally quantified variables as parameterized constants.

Table 4.2 shows the properties for the MUTEX controller. The properties MP1 and MP2 only refer to the controller variables. The remaining properties refer to both the controller variables and to the states of the threads. The properties MP3–7 are for concrete specifications and refer to concrete thread states. The properties MP8–12 are for the instances with counting abstraction and refer to the integer variables `IDLE` and `LOCKED` which represent the number of threads in the interface state `IDLE` and `LOCKED`, respectively.

Table 4.3 shows the properties we have verified using ALV on the rest of the concurrency controllers that we have used in the Concurrent Editor and that are in-

Table 4.2: Properties of the MUTEX controller

MP1	$AG(busy \Rightarrow AF(\neg busy))$
MP2	$AG(\neg busy \Rightarrow AF(busy))$
MP3	$AG(pc = LOCKED \Rightarrow AF(pc = IDLE))$
MP4	$AG(\neg(pc1 = LOCKED \ \wedge \ pc2 = LOCKED))$
MP5	$AG(pc = LOCKED \Rightarrow busy)$
MP6	$AG(\neg busy \Rightarrow pc = IDLE)$
MP7	$AG(\neg(pc1 = LOCKED \ \wedge \ pc2 = LOCKED))$
MP8	$AG(LOCKED > 0 \Rightarrow busy)$
MP9	$AG(\neg busy \Rightarrow IDLE > 0)$
MP10	$AG(LOCKED > 0 \Rightarrow AF(IDLE > 0))$
MP11	$AG(!(LOCKED > 2))$
MP12	$AG(LOCKED \geq 0 \wedge LOCKED \leq 1)$

troduced in Chapter 2. The concurrency controllers given in this table are as follows. AIRPORT is the concurrency controller for an Airport Ground Traffic Control simulation program mentioned in Section 2.1.2, BB is a bounded buffer, BB-MUTEX is a bounded buffer with a mutex lock, BB-RW is a bounded buffer with a reader-writer lock, and BARRIER is a controller for barrier synchronization. Note that, the controller BB-MUTEX should also satisfy the properties of MUTEX. The property given in the Table 4.3 is an additional property tailored for BB-MUTEX. Similarly, BB-RW should satisfy the properties of RW and the two additional properties given in this table.

Table 4.3: Properties of the remaining concurrency controllers

Controller	Property
BB	$AG(count \geq 0 \wedge count \leq size)$
BB-MUTEX	$\forall x\ AG((count = x \wedge LOCKED > 0) \Rightarrow AX(count = x))$
BARRIER	$\forall x\ AG((HUNG = x \wedge count < limit) \Rightarrow AX(HUNG \geq x))$
BB-RW	$\forall x\ AG((nR > 0 \wedge count = x) \Rightarrow AX(count = x))$
BB-RW-2	$\forall x\ AG(\neg busy \wedge count = x \Rightarrow AX(count \neq x \Rightarrow busy))$
AIRPORT	$AG(numRW16R \leq 1 \wedge numRW16L \leq 1)$
AIRPORT-2	$AG(numC3 \leq 1)$

Behavior Verification Results

We applied the presented behavior verification technique with ALV for the concurrency controllers introduced in Chapter 2 and for the concurrency controllers used in the Concurrent Editor and TSAFE. Table 4.4 shows the ALV performance for two generated instances of each these concurrency controllers: one concrete instance with 2 threads and one parameterized instance using counting abstraction. The verification results for the parameterized instances are stronger compared to the concrete cases since they indicate that the verified properties hold for arbitrary number of threads. We verified all of the associated properties given above with ALV for each concrete and parameterized concurrency controller instance. In this table, the columns Time and Memory show the time and memory consumed for the verification of concrete instances without any faults. The last two columns (P-Time and P-Memory) show the time and memory consumed for the verification of the

Table 4.4: Behavior verification performance of concurrency controllers for concrete instances with 2 threads and for parameterized instances

Instance	Time (s)	Memory(MB)	P-Time(s)	P-Memory(MB)
RW	0.17	1.03	8.10	12.05
MUTEX	0.01	0.23	0.98	0.03
BARRIER	0.01	0.64	0.01	0.50
BB-RW	0.13	6.76	0.63	10.80
BB-MUTEX	0.63	1.99	2.05	6.47
AIRPORT	0.51	45.57	0.83	61.50

parameterized instances.

In these experiments, the behavior verification took a small amount of time and used a fraction of the memory. The reason is that the Action Language specifications generated from the concurrency controllers are quite simple. Note that, there is no model extraction to generate these specifications since the `Controller` classes encode the concurrency behavior explicitly, hence serve a model for the synchronization policy. Therefore, there are no irrelevant details in the generated Action Language specifications. Moreover, there are no environment generation for the concurrency controllers because of the `ControllerStateMachine` classes. The developer has implemented the most general acceptable call sequences to the concurrency controller, which is the environment of the concurrency controller.

4.3.2 Interface Verification

During interface verification, we check that the assumption of the behavior verification is satisfied by the user threads of the concurrency controllers: each thread that uses a controller obeys the interface of that controller. In Chapter 3, we have discussed the formalizations for this verification and the thread interface correctness concepts. The threads should follow the action orderings defined at the interfaces of the concurrency controllers and should access the shared data protected by these controllers at the allowed interface states specified in the data stubs. In the same Chapter, we have explained how to perform interface verification with JPF. When there is an interface violation, JPF outputs a counter-example execution trace that leads to the violation. We have also explained that such verification can be performed on each concurrent thread separately with the thread isolation techniques. In this section, we first illustrate the thread isolation on the Concurrent Editor and TSAFE. Then we show the interface verification results for each concurrent thread in both examples.

Both TSAFE and Concurrent Editor are comprised of a server node and a client node. First we present the thread isolation for TSAFE, and then we continue with the thread isolation for the Concurrent Editor.

The TSAFE's client node is a program that consists of a main thread and two implicitly created threads. The main thread instantiates the GUI objects and establishes RMI connection to the TSAFE server. The implicitly created threads are the event thread that handles GUI events and the RMI thread that handles incoming RMI events triggered by the server. The environment of the main thread are modeled only

GUI component stubs and a stub for the `java.rmi.Naming` class both of which are provided by our framework as generic stubs. The event thread is isolated with GUI component stubs and an automatically generated driver (see Section 3.4 in Chapter 3). This driver instantiates GUI component stubs and creates all possible GUI events to the event thread. Since TSAFE's client has a large number of GUI components and JPF exhausts memory for all possible event lengths, this driver only generates all possible GUI event sequences of length 2. The client node has 2 RMI operations and 4 RMI events. These RMI operations and events are modeled with an automatically generated RMI stub and RMI driver (see Section 3.4 in Chapter 3). The generated driver registers an RMI stub of the TSAFE server node, and instantiates the client node. Then it produces all possible RMI events for all possible event sequences.

The TSAFE server is a program with two implicitly created threads, a main thread, and an explicitly created thread. The implicitly created threads are the RMI threads and the event thread. The explicitly created thread is the feed parser thread which reads messages from a socket and updates the flight database. The main thread creates a set of GUI components and instantiates the main application. The main thread does not launch the actual TSAFE application. The launching is done by clicking a *Launch* button in the GUI. Only after this click event an RMI connection and a feed socket is opened. In other words, the event thread performs the launch. Since this task does not involve concurrency, we have omitted these operations while creating the environment of the event thread. The other responsibility of the event thread in the server node is to handle the events created by a timer. Whenever a timer event occurs, all of the registered TSAFE clients are notified. Therefore, the

event thread driver first finds the `Timer` object, and then calls its registered listener in an infinite loop. The other explicit threads are the RMI threads, one per client. While modeling an RMI thread, we have modified the generated driver due to the launch mechanism in the server component and our objective of not modifying the application code. We have inserted a code that finds the launch button and sends a click event into the RMI driver. Finally, the feed parser thread is isolated with thread creation operation stubs and socket operation stubs. The feed parser thread is created at launch by the event thread and uses TCP sockets to get data supplied by an external feed source. We have modeled this external feed source by applying the TCP modeling methodology (see Section 3.4 in Chapter 3).

The client node of Concurrent Editor contains a main thread, two implicit threads, and explicitly created worker threads. The main thread does not modify any shared data. This thread only instantiates the client application and opens an RMI connection. The main thread is isolated with GUI component stubs and RMI connection stubs, similar to the isolation of the main threads above. The isolation of the event thread is achieved with a driver that produces all possible GUI events for the event sequence of length at most 2. The RMI thread is responsible for serving 5 different RMI event types. The RMI driver produces all possible event sequences with these event types. The rest of the threads are worker threads. Note that these threads share the same class definition. Therefore, we can perform interface verification on one worker instance and generalize the result for all worker threads since we reflect the influences of other threads with isolation process. The worker threads are isolated with thread creation operation stubs and concurrency controller interfaces which are

used for isolation of all threads.

The server node of Concurrent Editor has only a main thread and an RMI thread per client. The isolation of these threads is similar to the preceding discussion. The RMI thread, however, is different from the other RMI threads discussed above. This RMI thread serves the RMI events for two types of remote objects. One is the server object in the collaboration framework used. The remote call to this object creates 7 kinds of RMI events. The other one is the remote RW controller. The remote call to this object creates 4 kinds of RMI events.

After the thread isolation, we performed interface verification on each thread separately. Table 4.5 shows the JPF performance for interface verification of each isolated thread. The first 3 columns show the results for Concurrent Editor threads and the last 3 columns show the results for TSAFE threads. The threads of Concurrent Editor are denoted with the prefix CE and the threads of TSAFE are denoted with the prefix T. The server threads are labeled with Server and the client threads are labeled with Client. For example TServer-Feed denotes the feed parser thread at the server node of TSAFE.

During the interface verification of Concurrent Editor we have discovered several interface violation errors. One group of these errors was caused by not calling the correct controller method before accessing the shared data. Another group of errors was a violation of the controller call sequence because of the incorrectly handled exception blocks. This experiment shows the necessity of the interface verification and also shows that such errors can be captured in a realistic system such as the Concurrent Editor. The interface violations on TSAFE experiments will be discussed in

Table 4.5: Interface verification performance with thread isolation

Concurrent Editor			TSAFE		
Node-Thread	Time(s)	Memory(MB)	Node-Thread	Time(s)	Memory(MB)
CEServer-Main	2.77	3.20	TServer-Main	67.72	17.08
CEServer-RMI	185.85	67.14	TServer-RMI	91.79	20.31
CEClient-Main	2.92	3.87	TServer-Event	6.57	10.95
CEClient-RMI	229.18	127.94	TServer-Feed	123.12	83.49
CEClient-Event	1636.62	139.48	TClient-Main	2.00	2.32
CEClient-Worker	19.47	5.36	TClient-RMI	17.06	40.96
			TClient-Event	663.21	33.09

Section 4.4. We will show our verification technique is scalable to even larger and real systems such as TSAFE.

To demonstrate the effectiveness of our modular verification approach, we verified two of the concurrency controllers with JPF without using controller interfaces as stubs. Table 4.6 shows two of these examples, BB-RW and AIRPORT (see Section 2.1). The interfaces of these controllers are given in Figure 2.6 in Chapter 2. For this experiment, the user threads are kept simple. The threads execute concurrency controller actions and invoke the shared stub methods in an infinite loop. There are no other computations. In this experiment, the threads obey the controller interfaces. We used both depth-first and breadth-first search heuristics of JPF. The column labeled TN-S shows the number of user threads and the buffer size (for BB-RW). The memory usage is shown in the column labeled M, and the execution time is displayed in the column labeled T. For the BB-RW case it is not possible to verify the original

specification using a program checker such as JPF since the size of the buffer is an unspecified constant. To evaluate the performance of JPF we picked a fixed buffer size in the experiments reported in Table 4.6. JPF runs out of (512MB) memory (denoted by ↑) for buffer size 5 and 2 user threads for both heuristics because of the state space explosion. For the AIRPORT case the performance of JPF drops dramatically when the number of user threads increases because of the increase in the number of possible interleavings. JPF runs out of 512MB memory for 4 user threads for this example. On the other hand, the presented interface verification for the very same threads is performed successfully without exhausting the memory. The threads of BB-RW are verified in 2.74 seconds and used 1.43 MB memory. The threads of AIRPORT are verified in 3.33 seconds and used 2.15 MB memory. When using controller interfaces as stubs and thread isolation, the reachable state spaces become finite and we do not have to consider all possible thread interleavings. Therefore, we have achieved a significant state space reduction; thus, a dramatic improvement in the efficiency of the interface verification.

4.3.3 Finding the Shared Objects

The verification approach presented in this chapter relies on the assumption that the programmer knows all of the shared data that are accessed by more than one thread and protects them with a concurrency controller. When the programmer fails to identify all of the shared data, the verification results will not be helpful. Similar errors happen in standard Java programming when programmers do not use the Java synchronization primitives to protect access to shared objects. While reengineering

Table 4.6: Performance of JPF without using interfaces as stubs

		DFS		BFS	
Instance	TN-S	T(s)	M(MB)	T(s)	M(MB)
BB-RW	2 – 1	23.65	23.81	48.67	17.11
BB-RW	2 – 2	63.89	57.53	161.11	55.58
BB-RW	2 – 3	174.35	150.11	505.76	141.28
BB-RW	2 – 4	520.71	333.06	1542.68	329.13
BB-RW	2 – 5	↑	↑	↑	↑
BB-RW	3 – 2	↑	↑	↑	↑
AIRPORT	2	25.79	24.61	57.39	23.59
AIRPORT	3	1430.44	329.71	880.37	309.71
AIRPORT	4	↑	↑	↑	↑

TSAFE we found such an error where a shared variable used for RMI connection was not synchronized. We fixed this error by introducing a mutex controller. Such situations can be handled with an escape analysis technique. Escape analysis techniques are used to identify the objects which escape from a thread's scope and become accessible by another thread. Such analysis can be used to identify the objects which need to be protected.

Consider the following example with two threads sharing a `DataBuffer` protected by a BB-RW controller. (The concurrency controller and the data buffer implementations are given in Section 2.1.1.)

```
class Th1 extends Thread{
  private DataBuffer buffer;
  private BBRWInterface controller;
  ....
```

```
  public void run(){
    ....
    java.lang.Object obj=new Object();
    controller.w_enter_produce();
    buffer.put(obj);
    controller.w_exit();
    ....
  }
}
class Th2 extends Thread{
  private DataBuffer buffer;
  private BBRWInterface controller;
  ....
  public void run(){
    ....
    controller.w_enter_consume();
    Object obj=buffer.take();
    controller.w_exit();
    ....
  }
}
class MainClass {
  ....
  public static void main(String args[]){
    ....
    DataBuffer buffer=new DataBufferImpl(size);
    BBRWInterface controller=new BBRWController();
    ....
    t1=new T1(buffer, controller);
    t2=new T2(buffer, controller);
    t1.start(); t2.start();
  }
}
```

In this example, the object pointed by `obj` is shared and the programmer has failed to protect it by a concurrency controller. (This situation is explained in the next discussion.) The escape analysis tools we use identified the following objects that may escape from a thread's scope: 1) the `DataBufferImpl` instance created by the main thread, including the internal data array used in the implementation

of this buffer and the argument of the `insert` method of this object, 2) the controller `BBRWController`, including the `Action` instances, the `GuardedCommand` instances, exception objects, and the vector holding the guarded commands, 3) the object created by the `Th1` instance in the `run` method, which is the same as the argument of `insert` method of the buffer, and 4) system objects that are instantiated once and only read by the threads such as `java.security.Permission` objects.

The escape analysis techniques we investigated so far [19, 60] either do not scale to programs as big as TSAFE and Concurrent Editor or identify too many objects as shared. We think that this is a promising direction for future research.

How to handle Collections

When the shared data are collections (such as a vector, set, or a buffer whose elements are also objects), using a concurrency controller to protect only the collection instance is not sufficient. The shared collections may cause other objects to become shared as well. When an object is inserted into a shared collection by one thread, another thread can get the same object from that collection. The example presented in the preceding discussion shows an example for this situation. Because of the shared collection `DataBuffer`, `obj` becomes a shared data.

The objects that become shared through a collection should also be protected. One solution is to protect them with new controller instances, use shared stubs to define the access rules, and use the stubs *only* for these instances during interface verification. For the above example, we implement an `ObjectStub` that has one stub method for every method of `java.lang.Object` that asserts when the

method can be executed. This stub class corresponds to `SharedStub` in Figure 2.2. Finally, before interface verification, we change the creation of `obj` statement executed by `t1` to `obj=new DateStub(controller);` without changing the other `java.lang.Object` instantiations. In fact, this procedure is followed for all shared data classes where not all of their instantiations are shared by multiple threads.

4.4 Experimental Study with Fault Seeding

We conducted an experimental study with the goal of investigating the effectiveness of the presented DFV approach on safety critical air traffic control software. For this study we used the reengineered TSAFE implementation explained in Section 4.2. We created 40 new versions of the TSAFE source code by fault seeding. The seeded faults were created to resemble the possible errors that can arise in using the concurrency controller pattern such as making an error while writing a guarded command or forgetting to call a concurrency controller method before accessing shared data. We used both infinite and finite state verification techniques for detecting these faults based on our modular verification strategy supported by the concurrency controller pattern. The experimental study demonstrated the effectiveness of this modular verification strategy. During this experimental study we also developed a classification of the faults that can be found using the framework presented in this chapter.

During the experimental study we performed on TSAFE, we have collaborated with the developers of TSAFE testbed and formed two teams: 1) The University of California at Santa Barbara (UCSB) team which consists of the developers of

the presented verification technology and 2) The Fraunhofer Center for Experimental Engineering, Maryland (FC-MD) team which consists of the developers of the TSAFE testbed.

Experiment Setup

First, the UCSB team reengineered the TSAFE software as described in Section 4.2 and generated the drivers and the stubs for thread isolation as explained in Section 4.3.2. The reengineering of the TSAFE software using the concurrency controllers was done in 8 hours by the author of this dissertation (5.5 hours for the server component and 2.5 hours for the client component).

The UCSB team sent the reengineered TSAFE code to the FC-MD team. The FC-MD team created modified versions of TSAFE using fault seeding. The FC-MD team created two types of faults: *controller faults* were created by modifying the controller classes and *interface faults* were created by modifying the order of the calls to the methods of the controller classes. Each modified version contained either no faults, or one controller fault, or one interface fault, or one controller and one interface fault.

There are four types of controller faults: 1) *initialization faults* (CI) which were created by modifying the initialization statements in the controller classes, 2) *guard faults* (CG) which were created by modifying a guard in a guarded command, 3) *update faults* (CU) which were created by modifying an assignment in a guarded command, 4) *blocking/nonblocking faults* (CB) which were created by making a nonblocking action blocking or visa versa.

Table 4.7: Faulty versions

Fault Type	TSAFE Versions
CI	v2, v4
CG	v3, v6
CU	v7, v13, v14, v16, v24, v25
CB	v5, v21, v28, v34
IM	v7, v8, v10, v11, v15, v22, v23, v29
ICV	v1, v26, v27, v30, v31, v32, v33, v35–40
ICN	v12, v17, v18, v19, v20

Interface faults are categorized as: 1) *modified-call faults* (IM) which were generated by removing, adding or swapping calls to the methods of the controllers. 2) *conditional-call faults* which were generated by adding a branch condition in front of a method call to a controller. The conditional-call faults are further categorized as: a) *program-variable faults* (ICV) in which the created branch conditions used existing program variables, b) *new-variable faults* (ICN) in which the created branch conditions used new variables that were declared during fault creation.

After the fault seeding, the FC-MD team sent the modified versions back to the UCSB team. Table 4.7 shows the fault distribution for the forty modified versions of TSAFE (v1–40). The modified version v9 did not contain any faults. The UCSB team did not know the faults and which types of faults were in which version (or if there was any fault in a version). However, the UCSB team knew about the fault classification.

Results and Discussion

We ran the experiments in three batches with 25 (v1–25), 10 (v26–35) and 5 (v36–40) modified versions. After the verification of each batch both teams discussed the results. This allowed us to improve the experimental setup during the study and also helped us identify and focus on the weaknesses of the verification techniques.

As shown in Table 4.7, there were a total of 14 controller faults and 26 interface faults in versions v1–40. When we verified the controllers in versions v1–40 with ALV we found 12 faults in the controllers. The faults that were not found by ALV were the faults in versions v5 and v13 which were spurious faults, i.e., they are modifications in the controller classes which do not cause any failures in the controller behavior. For example, the modification in v13 changed an assignment in the `w_exit` action of the RW controller from `busy=false` to `busy=!busy`. However, this modification does not cause any failures since `busy` is always `true` when `w_exit` is called. The modification in v5 changed the `release` action in the MUTEX controller from blocking to nonblocking. Again this modification does not change the behavior of the controller since the guard of the `release` action is `true`, i.e., it never blocks.

In this experiment, we generated three instances of RW and MUTEX controllers for behavior verification: two concrete instances with 8 and 16 threads and a parameterized instance using counting abstraction. The verification and falsification of these instances are displayed in Table 4.8. The instances are suffixed with 8, 16, and P in the table to denote the concrete instances with 8 and 16 threads, and the parameterized instance, respectively. For the MUTEX controller we checked 7 properties

Table 4.8: Verification and falsification performance for the concurrency controllers of TSAFE

Controller Instance	Verify		Falsify	
	M(MB)	T(s)	M(MB)	T(s)
RW-8	6.36	2.36	3.26	0.34
RW-16	24.13	27.41	10.04	1.61
RW-P	12.05	8.10	5.03	1.51
MUTEX-8	0.41	0.02	0.19	0.02
MUTEX-16	1.08	0.05	0.54	0.04
MUTEX-P	0.98	0.03	0.70	0.12

on the concrete instances (MP1–MP7 in Table 4.2) and 7 properties on the parameterized instance (MP1, MP2 and MP8–12 in Table 4.2). For the RW controller we checked 10 properties on the concrete instances (RP1–10 in Table 4.1) and 11 properties on the parameterized instance (RP1–4 and RP11–17 in Table 4.1). Both verification and falsification of the MUTEX controller is more efficient compared to RW controller since it is a smaller specification with less number of variables. For the parameterized instances, the performances of both verification and falsification are typically between the concrete cases with 8 and 16 threads. Note that, even the verification results for the parameterized instances are stronger compared to the concrete cases, for falsification the results of the concrete and parameterized instances are equivalent. Both of them generate a counter-example behavior demonstrating the fault. It is possible for the concrete instances to miss a fault. However, in our experiments we did not observe this. Every fault that was found by the parameterized

instance of a controller was also found by the instance with 8 threads. Hence, our experiments indicate that concrete instances can be used for efficient and effective debugging of the controller behavior. After eliminating all the faults by the concrete instances, one could use the parameterized instances to guarantee correct behavior for arbitrary number of threads.

Among the 26 interface faults, interface verification using JPF identified 21 of them. Two of the faults (v22 and v33) that were not caught by JPF were spurious faults. However, the faults in versions v18, v19, and v20 were real faults which can cause failures but were not found by JPF. (We will discuss these faults in detail below.) The verification performance of JPF for interface verification of each thread of TSAFE is given on the rightmost three columns of Table 4.5 and the falsification performance results are shown in Table 4.9. Main threads do not have access to any controllers or shared objects so they cannot have any synchronization faults. We still list the verification time for the main threads to indicate the time it takes JPF to cover their state space. Typically falsification time with JPF is better than the verification time. This is expected since in the presence of faults JPF quits after finding the first fault without covering the whole state space. However, in some of the instances, JPF consumed more resources for falsification since the inserted faults either caused the execution of a piece of code which was not executed otherwise, or created new dependencies which increased the range of values used in the environment. Still, overall, falsification performance is better than the verification performance, especially for the more challenging verification tasks such as the TClient-Event thread and the TServer-Feed thread.

Table 4.9: Thread falsification performance

Node-Thread	M(MB)	T(s)
TClient-Main	–	–
TClient-Event	12.2	15.63
TClient-RMI	42.64	10.12
TServer-Main	–	–
TServer-Event	9.56	6.88
TServer-RMI	24.74	29.43
TServer-Feed	94.72	18.51

Fault Categorization: One of the outcomes of this experimental study was a clarification of the types of faults that can be verified using the presented approach. For example, during behavior verification we only check for errors in the initialization statements, guards, updates and blocking/nonblocking declarations. If a programmer changes the predefined helper classes (such as the Action class) and makes an error, the presented approach cannot find such an error. However, such errors can be avoided by using the automated optimization step since that step only uses the initialization statements, guards, updates and blocking/nonblocking declarations, i.e., the parts verified during behavior verification.

Completeness of the Controller Properties: Another problem we identified during the experimental study was the difficulty of listing all the properties that are relevant to the behavior of a controller. The initial set of properties we had about concurrency controllers was all about the controller variables did not relate the interface states to the variables of the controllers. During the experimental study we

quickly realized that we needed to specify more properties to find all faults that can be introduced. Eventually, the set of properties we identified found all the seeded faults; however, they are not guaranteed to find all possible faults. Our experience in this experimental study suggests that one could test the completeness of a set of properties for a controller by inserting faults to the controller and checking the modified controller with respect to the specified properties as we did in this experimental study. This is similar to mutation testing for measuring the effectiveness of a test set.

Difficulty of Finding Deep Faults: Finally, we would like to discuss the only real faults that were missed by the presented verification approach: the interface faults in versions v18, v19, and v20. The versions v17, v18, v19, and v20 were all created by adding a branch condition in front of a method call to a controller. The added branch condition tests if the value of a variable is less than a constant. If not, the call to the controller method is skipped. The variable in the branch condition is initialized to zero and is incremented every time the control reaches the inserted branch condition. The only difference between the faults in versions v17, v18, v19, and v20 was the constant value in the branch conditions which was 100, 1000, 10000, and 100000, for versions v17, v18, v19, and v20, respectively. Interface verification with JPF identifies the fault in v17 however misses the faults in v18, v19, v20. Clearly, these are convoluted faults. This fault type was suggested by the UCSB team as a way to challenge the interface verification step. These faults demonstrate that there is a limit to the depth of the faults that can be identified using explicit state verification techniques without running out of memory. In order to deal with this type of faults

symbolic analysis of the branch conditions may be necessary.

Thread Isolation: When we automatically isolate threads by generating environment models which allow maximum amount of nondeterminism, JPF runs out of memory. The user needs to provide some guidance in limiting the input domains and the input length. The dependency analysis we used was crucial for this task. Without dependency analysis it is not possible to identify what part of the input may be relevant to the synchronization behavior. One can approach this problem also from the design for verification perspective by developing interfaces for threads. We use the controller interfaces to model the environments of the concurrency controllers and shared data. Similarly, one can think of using interfaces for modeling the environments of threads.

4.5 Related Work

Assume guarantee style verification of software components has also been studied by Pasareanu et al. [85] in which LTL formulas are used to specify the environment (i.e., the interface) of a component. In our approach, on the other hand, the interface of a controller is a finite state machine. The controller behavior is verified assuming that the threads obey the finite state machines specifying the interface of that controller.

Model checking finite state abstractions of programs has been studied by several researchers [4, 25, 26, 37]. We present a modular verification approach where behavior and interface checking are separated based on the interface specification

provided by the programmer. Also, we use infinite state verification techniques for behavior verification instead of constructing finite state models via abstraction.

Yavuz-Kahveci et al. [105] specify concurrency controllers directly in Action Language. We eliminate the overhead of writing specifications in a specification language by introducing the concurrency controller pattern. Also, the authors do not address controller interfaces and interface verification to check the assumptions of concurrency controller behavior on the code.

Magee et al. [71] model concurrent programs as a finite state Labeled Transition System (LTS) written using a process algebra notation. The authors use a model checker called the Labeled Transition System Analyzer (LTSA) to verify the concurrent behavior model using finite state verification techniques. These techniques, however, cannot handle models with parameterized constants and unbounded variables and, most importantly, for arbitrary number of threads. Our approach can handle infinite-state concurrency controllers which can not be modeled using the LTSA approach. Note that the finite state machines we use to model the controller interfaces only specify how the controller should be used by user threads, not the controller behavior.

In Chapter 2 we have discussed related work regarding to the concurrency controller pattern. The verification aspects of these works are also related to the DFV approach presented here.

Chapter 5

Design for Verification of Asynchronously Communicating Web Services

This chapter presents a design for verification (DFV) approach for developing reliable web services based on the peer controller pattern. The focus is on composite web services that consist of asynchronously communicating peers. In this framework, our goal is to automatically verify properties of interactions among such peers. The peer controller design pattern eases the development of such web services and enables a modular, assume-guarantee style verification strategy. In this design pattern, each peer is associated with a behavioral interface description, which specifies how that peer interacts with other peers (see Section 2.2 in Chapter 2). The interface of a peer can be viewed as a contract between that peer and other peers which interact with it. These peer interfaces are finite state machines that are crucial for the verification approach presented in this chapter. In addition to verification, these peer interfaces are also used for automated BPEL specification generation to publish for

interoperability.

In this chapter we show that if the developers use the peer controller pattern, then by using model checking techniques, we can automatically check the properties of a composite web service (behavior verification) and the conformance of the peer implementations to their interfaces (interface verification). The interface verification is performed with Java PathFinder [99] by using peer interfaces as stubs for asynchronous communication components. This approach solves the environment generation problem in this context and enables verification of each peer in isolation, improving the efficiency of the interface verification significantly. The behavior verification is performed with the explicit and finite state model checker SPIN [58], which allows us to model asynchronous messaging. We check safety and liveness properties of the composite web service specification using the conversation model [46] (see Section 2.2.3).

Since SPIN is a finite state model checker, the size of the message queues needs to be bounded. Such bounded verification guarantees correctness only with respect to the set bounds. Moreover, model checking a composite web service that communicates asynchronously with unbounded queues is undecidable [46]. Note that, this is not just a theoretical problem. Asynchronous messaging is supported by messaging platforms such as Java Message Service (JMS) [64], Microsoft Message Queuing Service (MSMQ) [77], and Java API for XML Messaging (JAXM [63]) where the message queues are not bounded.

We adapt the *synchronizability analysis* proposed by Fu et al. [47] to our framework in order to verify properties of composite web services in the presence of

unbounded queues. A composite web service is called synchronizable if its conversation set does not change when asynchronous communication is replaced with synchronous communication. We check the sufficient conditions for synchronizability proposed by Fu et al. [47] based on the peer controller pattern. This automated synchronizability check enables us both to reason about the global behavior with respect to unbounded queues and to improve the efficiency of the behavior verification (by removing the message queues, and hence, reducing the sate space, without changing the conversation behavior).

In this chapter, after the presentation of the above modular verification approach, we show that hierarchical state machines (HSMs) can be used to specify the peer interfaces. HSMs provide a compact representation that reflects the natural hierarchy of the service behavior and can specify concurrent executions of operations. We discuss how we extend the presented modular verification approach, automated BPEL specification generation, and the synchronizability analysis to HSMs.

5.1 Composite Web Service Examples

To illustrate the DFV approach for web services based on the peer controller pattern, we use the following examples: a Loan Approval service, a book and CD ordering service, a Travel Agency service, and a Purchase Order Handling service.

Loan Approval service is explained in Section 2.2.1 in Chapter 2. A brief summary of this example is as follows. A customer requests a loan for some amount. If the amount is small, the loan request is approved. For large amounts, a risk as-

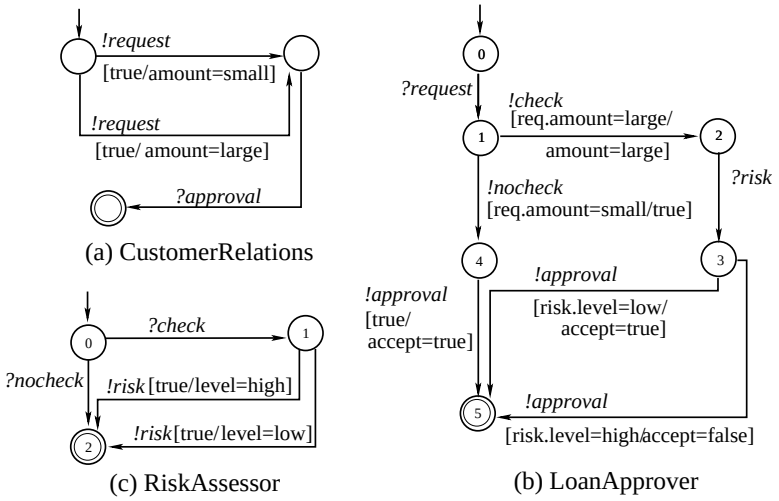
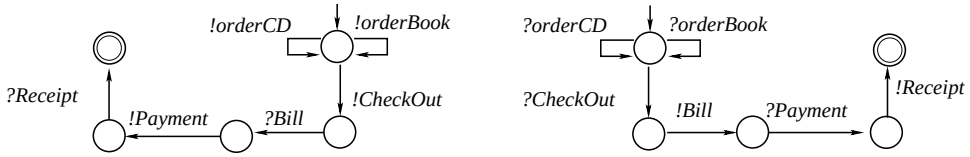


Figure 5.1: Peer interfaces of Loan Approval service

sessment service decides a risk level. The loan request is approved when the risk level is low and denied when the risk level is high. The Loan Approval service is composed of three peers: CustomerRelations through which customers make loan requests, LoanApprover that realizes the loan service, and RiskAssessor that calculates and provides risk information. The peer interfaces of these peers are given here in Figure 5.1 again for convenience.

The second example is a composite web service (BookCD Order) where a client can order several books and CDs that are delivered by a supplier service. The client peer places arbitrary number of CD and book orders. After ordering the products, the client requests a checkout. The supplier calculates the total price and sends a bill to the client. Client sends the payment and gets a receipt from the supplier. This example is composed of two peers, Client and Supplier peers. The messages that are

exchanged among these peers are: *orderBook*, *orderCD*, *CheckOut*, *Bill*, *Payment*, and *Receipt*. These messages have a number of fields (attributes); however, none of them is a control attribute, which influence the communication logic (i.e. the interface). The peer interfaces defining the contracts of these peers are shown in Figure 5.2.



(a) Client (b) Supplier

Figure 5.2: Peer interfaces of Book and CD Ordering service (BookCD Order)

The next example is the Travel Agency service. This service reserves a hotel, a car and a flight according to a request coming from a customer. A travel agent gets the travel information from the customer. The travel information is a number of dates and vacation destinations. After getting this information, the travel agent starts the accommodation and transportation reservations. These reservations are made concurrently and do not have any dependency between them. The transportation reservation books a car and a flight if the customer wants a transportation reservation. The bookings of the car and the flight also occur concurrently to complete the task in a timely manner. A flight reservation could be one-way or round-trip depending on the request. Finally, an itinerary is sent to the customer. Travel Agency is composed of five peers: Customer, TravelAgent, CarReserve, HotelReserve, and FlightReserve. The Customer peer provides the information about the customer

request to the TravelAgent with the *place, date, process, transportResv*, and *transportNoResv* messages. The TravelAgent arranges reservations according to the requests of the Customer and responds to the Customer with the *travelInv* message. The other three services perform specific reservation requests. The HotelReserve communicates using the *reqHotel*, and *hotelInv* messages, the CarReserve using the *reqCar*, and *carInv* messages and the FlightReserve using the *reqFlight*, and *flightInv* messages. Among these messages, only the *reqFlight* and *transportResv* have a control attribute, which influence the communication logic. The control attribute of *reqFlight* message is a boolean *roundTrip* field, and the control attribute of the *transportResv* message is an enumerated *flight* field whose domain is {*roundtrip, oneway*}. Note that, it is quite tedious to define the peer interfaces for this example with finite state machines because of the concurrent execution requirements. To specify the interfaces of these peers, we use HSMs which will be discussed in Section 5.5.

The last example is the Purchase Order Handling service described in the BPEL 1.1 specification [20]. In this example, a customer makes a purchase order to a vendor. The vendor calculates the price for the order including the shipping fee, arranges a shipment, and schedules the production and shipment. The vendor uses an invoicing service to calculate the price, a shipping service to arrange the shipment, and a scheduling service to handle scheduling. To respond to the customer in a timely manner, the vendor performs these three tasks concurrently while processing the purchase order. There are two control dependencies among these three tasks that the vendor needs to consider: The shipping price is required to complete the final price

calculation, and the shipping date is required to complete the scheduling. After these tasks are completed, the vendor sends the final invoice to the customer. The web service for this example is composed of five peers: CRelations, Purchasing, Shipping, Scheduling, and Invoicing. Customers order products using the CRelations peer. The CRelations peer communicates with the Purchasing peer which plays the role of the vendor described above. The Purchasing peer responds to the CRelations with a *replyOrder* message. The remaining services are the ones that the Purchasing peer uses to process the product order. The Shipping peer communicates with *reqShipping*, and *schedule* messages, the Scheduling peer with *productSchedule*, and *shippingSchedule* messages, and the Invoicing peer with *initialize, shippingPrice*, and *invoice* messages. None of these messages have control attributes that influence this communication logic. Similar to the Travel Agency example, we will use HSMs to define the peer interfaces for the Purchase Order Handling system due to the concurrent execution requirements and dependencies.

In all of the composite web services described in this section, the communication among the peers is through asynchronous messaging. These services can process more than one customer request at a time. Each customer request is processed according to the associated control logic described above.

5.2 Conversations and Synchronizability

While analyzing the interactions of peers in a composite web service, we use the conversation model [22, 46, 47] explained in Section 2.2.3 in Chapter 2. A conver-

sation is the sequence of messages exchanged among peers, recorded in the order they are sent. A conversation is complete if at the end of the execution each peer ends up in a final state and each message queue is empty. For simplicity, we call a complete conversation a "conversation" for the rest of this chapter. The notion of a conversation captures the global behavior of a composite web service where each peer executes correctly according to its interface specification, and every message ever sent is eventually consumed. It is reasonable to assume, based on the messaging frameworks provided by the industry [64, 77, 63], that no messages are lost during transmission.

In Chapter 3, we have defined conversations generated by a composite web service based on the peer interfaces. For example, one conversation generated by the Loan Approval system is as follows: *request(amount=large), check(amount=large), risk(level=high), approval(accept=false)*. (The control attribute of a message is written next to the message in parenthesis.) Another sample conversation is the following which is generated by the BookCD Order service. *orderCD, orderBook, orderCD, CheckOut, Bill, Payment, Receipt*. (The messages in the BookCD Order service have no control attributes as discussed in Section 5.1.)

Using the conversation model, we analyze the interactions of a composite web service by verifying properties on conversations generated by that web service through model checking techniques. Fu et al. [46] discuss the extension of temporal logic properties to specify properties of conversations. A composite web service satisfies an LTL property if all the conversations generated by the service satisfy the property.

Consider the two services that generate the above conversations. During the

execution of the Loan Approval service the input queue of each peer contains at most one message. However, this may not always be the case such as the case with BookCD Order service. The BookCD Order service specification has an infinite state space when the input message queues are not bounded. In fact, model checking conversations of a composite web service that communicates asynchronously with unbounded queues is undecidable [46].

This problem is addressed by Fu et al. [46, 47]. A technique called *synchronizability analysis* was developed to identify asynchronously communicating finite state machines (which are the peer interfaces of a composite web service in this context) which can be verified using finite state model checking techniques automatically. Fu et al. present a set of sufficient conditions on the control flows of these state machines, so that they generate the same set of conversations under both the synchronous and asynchronous communication semantics. Since LTL properties are defined over conversations, if these synchronizability conditions are satisfied, the verification results using synchronous semantics hold for the usual asynchronous semantics for web services. The composite web service is *synchronizable* if it satisfies these conditions. In other words, if a composite web service is synchronizable, we can model check the service with respect to conversation properties by replacing asynchronous communication (with unbounded message queues) with synchronous communication that does not change the conversation set generated by the composite web service. Note that, verification of a system which consists of synchronously communicating finite state machines is decidable since the state space of the composed system is finite. In fact, the state space of the composed system can

be constructed by taking the Cartesian product of the states of the individual state machines.

We adapt the synchronizability analysis to our framework based on the peer controller pattern in order to reason about interactions and verify conversation properties with respect to unbounded message queues. In the following sections, we discuss the synchronizability conditions and their realization in our framework in detail.

5.3 Verification of Composite Web Services

In this section, we present our modular and assume-guarantee style verification technique for a composite web service based on the peer controller pattern. Our goal is to automatically verify conversation properties and to analyze the interactions among the asynchronously communicating peers. The presented approach exploits the explicit peer interface definitions. The verification consists of two steps: 1) *Behavior verification*: Verification of the global behavior of the composite system using the conversation model assuming that the peers obey their peer interfaces. 2) *Interface verification*: Verification of the peer implementations to make sure that each peer conforms to its peer interface, which defines the order of messages the peer can send and receive. This verification strategy solves the environment generation problem for the peers and improves the efficiency of verification and hence, makes automated verification of realistic web services feasible. We use the Loan Approver service to illustrate the presented verification approach.

5.3.1 Behavior Verification

We use the model checker SPIN [58] during behavior verification of a composite web service written based on the peer controller pattern. The peer controller pattern, presented in Section 2.2, dictates explicit implementations of peer interfaces as state machines and implementations of message stubs that only preserve control attributes of a message instance. Since a peer interfaces define the behavior of a peer and since the peer implementations assumed to obey their interfaces, we can verify safety and liveness properties of a composite service by using only the peer interfaces. The behavior verification is based on the peer controller semantics and the projections discussed in Chapter 3, i.e., the transition system $T(W)$, where $W = (class(M), I_1, I_2, \ldots, I_k)$ is a composite web service specification with k peers and the peer interfaces $I_1, \ldots, I_k$, and the projection function Π_3 (see Section 3.2.2 and Section 3.3.4 for details).

SPIN is an explicit and finite state model checker for concurrent systems. The input to SPIN is a bounded model of a system written in Promela, which is the specification language of SPIN, and a set of LTL properties. SPIN focuses on proving the correctness of process interactions. Processes can interact with each other through shared variables and message passing through communication *channels*. Because of this *channel* structure, which is suitable for modeling the asynchronous messaging among the web services, we choose the model checker SPIN in behavior verification for composite web services.

We have implemented a translator that takes the peer interface implementations (corresponding to `CommunicatorInterface` in the pattern displayed in Figure

2.12) and the message stubs (corresponding to `MessageStub` in the pattern) as input, and automatically generates a specification in Promela. Here we explain this translation with the aid of Loan Approval service, which comprise of three peers. Given the message stub implementations and the three peer interface implementations, the translator generates a Promela specification with three process types; one process type for each peer interface. The following is an excerpt from the generated specification.

```
#define size 5
/*message names*/
mtype ={requestType, approvalType, nocheckType, checkType,riskType}

/*data domains*/
mtype ={undef1,small,large}//amount domain
mtype ={undef2,low,high}//level domain
...//other data domains

/*message types*/
typedef approval{ bool accept; }
typedef request{ mtype amount; }
...//other message types
message lastmsg;//holds the last message

/*channels*/
chan customerQ=[size] of {mtype,message}
chan approverQ=[size] of {mtype,message}
chan assessorQ=[size] of {mtype,message}

proctype LoanApprover(){
  short state=0;
  nocheck nocheckmsg; check checkmsg;
  approval  approvalmsg; risk riskmsg; request requestmsg;
  message msg;
  do
  ::state==0 ->
    if
    ::approverQ?[requestType,msg]->/*receive*/
      approverQ?requestType,msg;
      requestmsg.amount=msg.requestmsg.amount;
```

```
      state=1;
    fi
  ::...//other transitions
  ::state==3 ->
      if
      ::riskmsg.level==low ->/*guardV*//*send*/
        approvalmsg.accept=true;/*guardP*/
        msg.approvalmsg.accept=approvalmsg.accept;
        atomic{
             /*update function*/
         lastmsg.approvalmsg.accept=approvalmsg.accept;
         customerQ!approvalType,msg; }
        state=5;
      ::riskmsg.level==high ->/*guardV*//*send*/
        approvalmsg.accept=false;/*guardP*/
        msg.approvalmsg.accept=approvalmsg.accept;
        atomic{
             /*update function*/
         lastmsg.approvalmsg.accept=approvalmsg.accept;
         customerQ!approvalType,msg; }
        state=5;
      fi
  ::state==5 ->break; /*final state*/
  od;
}
proctype CustomerRelations(){...}
proctype RiskAssessor(){...}
init{...}
```

The first part of this specification declares constants, types and global channels. The message name domain is defined with `mtype` which is the enumerated type in Promela. The domains of the control attributes are defined similarly. The message types are declared as type constructs (`typedef`) holding the values of control attributes. The global variable `lastmsg` holds the last message transmitted. This variable is of type `message` which combines all the message types defined. The global channel variables are the asynchronous communication channels simulating the input message queues of the peers. In this example, the size of the channels is re-

stricted to 5, which is given as an input to the specification generator. These channels are defined to carry elements consisting of a message name and a `message`.

The second part is a set of process type definitions. In Promela, the `proctype` keyword is used for defining a concurrent process. One concurrent process definition is generated per peer. These definitions are used for defining the behavior of a peer by implementing the state machine specified by the peer interface `CommunicationInterface`. In the generated Promela code, each process definition has a local variable named `state`. This variable holds the current state of the state machine. A process also has one local variable for each message type it sends or receives. The body of a process is a single loop which nondeterministically chooses an operation to execute depending on the `state`. At each state, there is a conditional selection of sending, receiving and terminating operations. At verification time, one of the operations whose enabling condition is true is selected nondeterministically. The last part is the `init` block which instantiates the concurrent processes.

Let us consider the process type `LoanApprover`. The body of this process encodes the finite state machine given in Figure 5.1(b). This excerpt shows one example for each of the receiving, sending and terminating operations in that order. State 5 is a final state of the loan approver peer. Therefore, when the `state` is 5, the loop is terminated. When the `state` is 0, the process has a receive choice. If there is a *request* type message in the approver queue, the receive operation is enabled. This condition is implemented with `approverQ?[requestType,msg]` statement which does not alter the queue contents. When this condition holds, the request message is taken from the queue, the message contents are stored in the local

request message variable, and the state is updated. This fragment corresponds to the transition from state 0 to state 1 with *?request* in Figure 5.1(b).

When the `state` is 3 the process has two send choices. The first choice corresponds to the send transition with an *approval* message class and a guarding condition $g = g_v \wedge g_p$ where $g_v \equiv v_{\texttt{risk.level}} = \texttt{low}$ and $g_p \equiv$`accept=true`. This guarding condition is described in the communication stub of the loan approver peer as `last_risk.level()==low` for guardV and as `approval.accept() ==true` for guardP (see Section 2.2.4). This Promela code fragment checks the risk level, sets the `accept` field of the approval message to true, and constructs a message to be sent. Next, atomically `lastmsg` is updated and the approval message is sent. Finally, the state is updated. The second choice corresponds to the send transition with *approval* message class where the guarding condition is $g_v \equiv v_{\texttt{risk.level}} = \texttt{high}$ and $g_p \equiv$ `accept=false`.

Using this automatically generated specification, we can check the LTL properties about the global behavior of the composite web service using the conversation model. The set of atomic properties are the predicates on the messages. LTL properties are constructed from atomic properties, boolean logic operators ($\wedge$, $\vee$, $\neg$) and temporal logic operators (G: globally, F: eventually, U: until). An example property for the Loan Approval service is as follows: "Whenever a request message with a large amount is sent, eventually an approval message (with accept or reject) will be sent."

SPIN is an explicit state model checker, and therefore, the sizes of the channels need to be bounded. For example, in the above Promela specification the sizes of

the channels are bounded with the `size` constant. Bounding the sizes of the communication channels, however, poses a problem since the verification results only hold as long as the channel sizes remain within the set bounds. Next, we address this problem.

Behavior Verification with Unbounded Message Queues

Bounded verification using SPIN can only give developers a certain level of confidence. It cannot ensure freedom of bugs with respect to the LTL properties we are considering. While the general problem of modeling checking a composite web service that communicates asynchronously with unbounded queues is undecidable [46, 47], as noted in Section 5.2, we can identify the services that can be verified in the presence of unbounded queues with synchronizability analysis. If a composite web service is synchronizable, the service can be verified by replacing asynchronous communication (with unbounded message queues) with synchronous communication without changing the conversation set [45].

Synchronizability analysis uses two sufficient conditions to restrict control flows of the state machines: 1) synchronous compatibility and 2) autonomous condition. Synchronous compatibility condition requires that, if we construct a state machine which is the Cartesian product of the peer interfaces, there should not be a state in the product machine in which one peer is ready to send a message, but the receiver for that message is not in a state where it can receive it. The autonomous condition requires that at any state, a peer has exactly one of the following three choices: 1) to send, 2) to receive, or 3) to terminate. Note that the autonomous condition still

allows nondeterminism. A peer can chose which message to send nondeterministically.

We have implemented these sufficient conditions for synchronizability based on the peer controller pattern. Given the peer interface implementations with `CommunicationInterface` classes, we automatically check the synchronizability of the composite service. If the composition is synchronizable, the Promela code with synchronous communication is generated. Otherwise, the reason of the condition violations is displayed. Note that, when the synchronizability conditions are not met, bounded verification can still be used.

The Promela code generated for a synchronizable service has two differences from the Promela code given above. First, the queue (channel) size is fixed to 0 which in Promela means that processes should synchronize when exchanging messages. The other difference is the implementation of the receive operations. Instead of inquiring the queue contents, the messages are received first and the appropriate action is performed depending on the message type. We need this modification because when the channel size is 0, the channels do not store messages. Therefore the inquiry `peerQ?[msgtype, msg]` always returns false. Below, we give parts of the Promela specification generated from the peer interfaces of the Loan Approval service for synchronous communication illustrating the difference from the previous Promela code.

```
...//global definitions

/*channels*/
chan customerQ=[0] of {mtype,message}
chan approverQ=[0] of {mtype,message}
chan assessorQ=[0] of {mtype,message}
```

```
proctype LoanApprover(){
  ...// local definitions

   do
  ::state==0 ->
    if
    ::approverQ?msgtype,msg; /*receive*/
       if
       ::msgtype==requestType ->
         requestmsg.amount=msg.requestmsg.amount;
         requestmsg.fortest=msg.requestmsg.fortest;
         state=1;
       fi;
    fi
  ::...//other transitions
  ::state==5 ->break; /*final state*/
  od;
}
proctype CustomerRelations(){...}
proctype RiskAssessor(){...}
init{...}
```

With the aid of this automated synchronizability analysis, we can both reason about the global behavior with respect to unbounded queues and improve the efficiency of the behavior verification. Since the messages are not buffered, the state space of the specification is reduced that can lead to a significant improvement in the behavior verification.

5.3.2 Interface Verification

The goal of interface verification is to make sure that each peer implementation conforms to its peer interface, which defines the order of messages the peer can send and receive. In the loan approval service, for example, an approver thread should not send a *check* message to the RiskAssessor before getting a loan request

with a large amount. In other words, during interface verification, we check if the assumption of the behavior verification is satisfied.

In Chapter 3, we have discussed the formalizations for the interface verification and the interface correctness criterion. In the same Chapter, we have explained how to perform interface verification with JPF by using `CommunicationInterface` classes defining the peer interfaces via the provided `StateMachine` class. For each peer, we use its peer interface implementation as a stub for communication controller that realizes the asynchronous communication mechanism. Since these peer interfaces are finite state machines and abstract the asynchronous messaging with other peers, the efficiency of the interface verification is improved significantly.

In addition to verifying each peer implementation separately, we further improve the efficiency of the interface verification by checking a peer implementation for one session. Since, in the peer controller pattern, each session is independent and does not affect other sessions, it is sufficient to check the peer implementations for one session.

To perform the interface verification, the communication controller and message instances are replaced with communication stub and message stubs by a source-to-source transformation. With this transformation the asynchronous communication mechanism, which cannot be handled by JPF, is abstracted away. However, we still need to write a small driver to instantiate the service. The reason is that JPF requires standalone programs as input, but a peer is a servlet which does not have a main method. This simple driver class contains only a main method that consists of three statements: 1) instantiating the communicator stub, 2) instantiating an application

thread, and 3) starting the application thread. After these steps, the resulting program is given to JPF to search for interface violations. Note that, using these stubs and the driver we close the environment of a peer. Our verification approach solves the environment generation problem and enables verification of each peer in isolation.

5.4 BPEL Generation

Given a composite web service whose peers are implemented based on the peer controller pattern, we generate BPEL specifications from the peer interfaces automatically. As discussed earlier, the peer interfaces are specified with finite state machines. In this section, we present our automated BPEL generation from peer interfaces.

Before creating BPEL files, our generator creates one WSDL with all message type definitions and one WSDL per peer defining its port type, partner link types and bindings. These WSDL specifications are used in BPEL files. Then, we create one BPEL file per peer. The specification contains partner links definitions to access other peers, variable declarations, and behavior description of the peer. In the variable declaration, one variable per message type is defined to store the last message content.

Here we give the mapping of send transitions and receive transitions to BPEL activities. Consider the transitions originating from a state. If these transitions are send transitions, the corresponding BPEL fragment consists of a `switch` clause which has one `case` for each different send transition. The condition of `case` corre-

sponds to the guardV expression (g_v) of guarding condition of a send transition. The inner activity of this `case` block contains an assignment which corresponds to the guardP (g_p) of the guarding condition, one invoke statement, and another assignment statement that updates the state variable, in this order.

Consider the send transition from state 3 to state 5 with the label *!approval* [risk.level=low/accept=true] shown in Figure 5.1(b). This is the send transition $(3,!\textit{approval}, g, u, 5)$ where u is the update function and $g = g_v \ \wedge g_p$ is the guard condition with $g_v \equiv v_{\texttt{risk.level}} = \texttt{low}$ and $g_p \equiv \texttt{accept} = \texttt{true}$ in the Loan Approver service. The code generated for this transition is:

```
<case condition="getVariableData('risk', 'level')='low'">
 <sequence>
  <assign> <copy>
     <from expression="'true'"/>
     <to variable="approval" part="accept"/>
  </copy> </assign>
  <invoke partnerLink="CustomerRelations"
     portType="ns1:CustomerRelationsPT"
     operation="ns1:approval"
     inputVariable="approval">
   </invoke>
  <assign> <copy>
     <from expression="'5'"/> <to variable="state"/>
  </copy> </assign>
 </sequence>
</case>
```

In the case of receive transitions originating from a state, there are two kinds of resulting BPEL code fragments. If there is a single receive transition, a `receive` statement is generated. For example, for the transition from state 2 to state 3 with the label *?risk* in the LoanApprover interface, the generated code is:

```
<sequence>
 <receive partnerLink="RiskAssesor"
   portType="ns3:RiskAssessorPT"
```

```
   operation="ns2:risk" variable="risk">
 </receive>
 <assign> <copy>
     <from expression="'3'"/> <to variable="state"/>
 </copy> </assign>
</sequence>
```

If there are multiple receive transitions, we use the `pick` construct which has one `onMessage` per receive transition. For example, the initial state (state 0) of Figure 5.1(c), defining the interface of Risk Assessor peer, has two outgoing receive transitions: one is targeted to state 1 with the label *?check*, and the other is targeted to state 2 with the label *?nocheck*. The generated code fragment is:

```
<pick>
 <onMessage partnerLink="LoanApprover"
   portType="ns2:LoanApproverPT"
   operation="ns3:check" variable="check">
    <assign> <copy>
     <from expression="'1'"/> <to variable="state"/>
    </copy> </assign>
 </onMessage>
 <onMessage partnerLink="LoanApprover"
   portType="ns2:LoanApproverPT"
   operation="ns3:nocheck" variable="nocheck">
    <assign> <copy>
     <from expression="'2'"/> <to variable="state"/>
    </copy> </assign>
 </onMessage>
</pick>
```

When the interface is nondeterministic, we generate abstract BPEL processes. There are two situations that requires nondeterminism: 1) when there exists both send and receive transitions originating from one state, 2) when the guarding conditions of the send transitions originating from one state are not disjoint. We create nondeterminism by adding an extra variable, using only *opaque* assignments to this

variable, and selecting the choices based on the value of this extra variable. According to the BPEL 1.1 specification, an opaque assignment to a variable sets a nondeterministic value chosen from the value space of the variable.

The following is an excerpt from the BPEL specification generated for the peer interface of LoanApprover given in Figure 5.1(a).

```
<process name="LoanApprover" ...>
 <partnerLinks> ...  </partnerLinks>
 <variables> ...
    <variable name="state" type="xsd:string"/>
    <variable name="exit" type="xsd:string"/>
 </variables>
  <sequence>
     ...<!--receive loan request, and
            set '1' to state and create and instance -->
   <assign>
      <copy><from expression="'no'"/> <to variable="exit"/></copy>
   </assign>
   <while condition="exit!='no">
     <switch>
      <case condition="state='1'">
        <sequence>
        ...<!--send check or nocheck message -->
        </sequence>
       </case>
      ...<!--other choices -->
      <case condition="state='5'">
         <assign><copy>
           <from expression="'yes'"/>
           <to variable="exit"/>
         </copy></assign>
      </case>
     </switch>
   </while>
  </sequence>
</process>
```

The specification contains two special variables. The variable `state` represents current state, and the variable `exit` is used for termination condition. The process

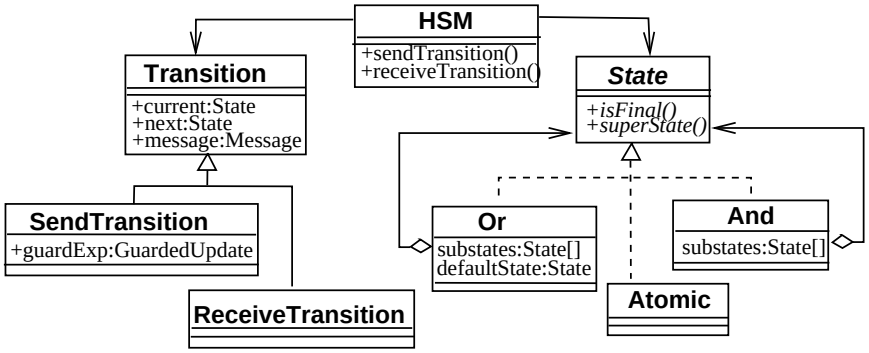


Figure 5.3: Hierarchical state machines

description is a `while` loop that terminates depending on the value of `exit`. The body of the loop selects a `case` block based to the value of `state`.

Each block contains a subactivity. If the interface state is specified as final, the local `exit` variable is set. Otherwise, the corresponding send or receive activities are performed whose code fragment construction is discussed above.

5.5 Verifiable Web Services with Hierarchical Interfaces

Finite state machines are powerful enough to specify behavioral interfaces of typical web services and they are suitable for automated reasoning. However, behavioral interfaces represented as finite state machines may contain a large number of states and may be hard to understand since they lack high-level structure. In this section, we propose using hierarchical state machines (HSMs) to specify the peer interfaces.

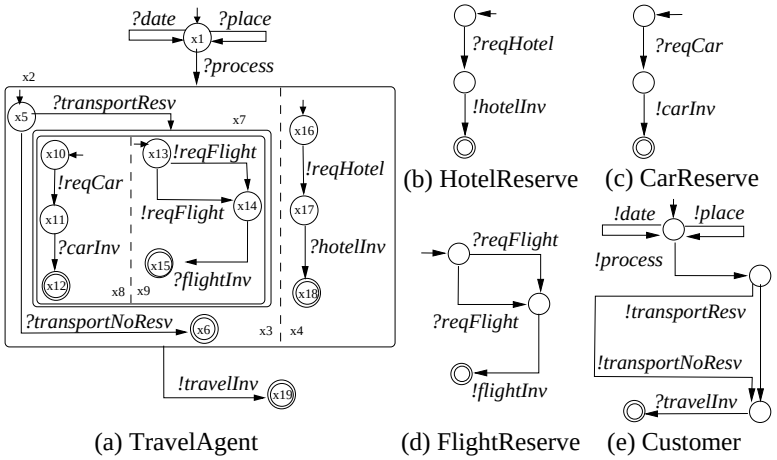


Figure 5.4: Peer interfaces for Travel Agency

With HSMs we achieve a compact representation that 1) reflects the natural hierarchy of the service behavior, 2) can specify concurrent executions of operations, and 3) is suitable for automated verification.

In this section, we will use the Travel Agency and Purchase Order Handling services for illustration. These examples are introduced in Section 5.1. The HSMs specifying the peer interfaces for the Travel Agency and the Purchase Order Handling services are shown in Figures 5.4 and 5.5, respectively.

The class diagram in Figure 5.3 shows the structure of the HSMs. An HSM state can either be an atomic state or a composite state with substates. There are two types of composite states: `And` and `Or` states. In Figures 5.4 and 5.5, composite states are shown as rectangles and atomic states are shown as circles. The substates of `And` states are separated using dashed lines. Each `Or` state has a unique default substate (shown with an arc without a source). Each state has an attribute called

`final` which denotes if it is a final state (final states are shown with double lines). We assume that each HSM has a *root* state at the top of the state hierarchy, which is an `Or` state (we do not show this root state in the figures).

The transitions of an HSM are partitioned into send and receive transitions similar to flat peer interfaces. Similar to send transitions of flat peer interfaces, the send transitions of HSMs have guarding conditions which are specified with the same syntax at the flat peer interfaces. Consider the Travel Agency service. Recall that, in this example, a flight reservation could be for a one-way or a round trip flight depending on the request of the customer. To realize this requirement, the TravelAgent peer has two send transitions that correspond to these two cases. One of these send transitions is $(x_{13}, !\textit{reqFlight}, g^1, u^1, x_{14})$ with the guarding condition $g^1 = g_v^1 \wedge g_p^1$ where $g_v^1 \equiv v_{\texttt{transportResv.flight}} = \texttt{oneway}$, $g_p^1 \equiv$`roundTrip = false`, $attr(\textit{reqFlight}) = \{\texttt{roundTrip}\}$, $attr(\textit{transportResv}) = \{\texttt{flight}\}$, and $v_{\texttt{transportResv.flight}}$ is an interface variable of the peer interface of the Travel Agent peer. The other send transition is $(x_{13}, !\textit{reqFlight}, g^2, u^2, x_{14})$ with the guarding condition $g^2 = g_v^2 \wedge g_p^2$ where $g_v^2 \equiv v_{\texttt{transportResv.flight}} =$`roundtrip` and $g_p^2 \equiv$`roundTrip = true`.

Finally, in HSMs, dependency arcs can connect states to transitions. A transition can only be taken if the current configuration of an HSM contains the states that are the sources of all the dependency arcs that point to that transition (formal description is given below). In Figure 5.5, there are two dependency arcs. One of them implies that the transition from a_9 to a_{10} can only be taken if the current configuration contains the states a_9 and a_6. The other dependency arc implies that the transition from

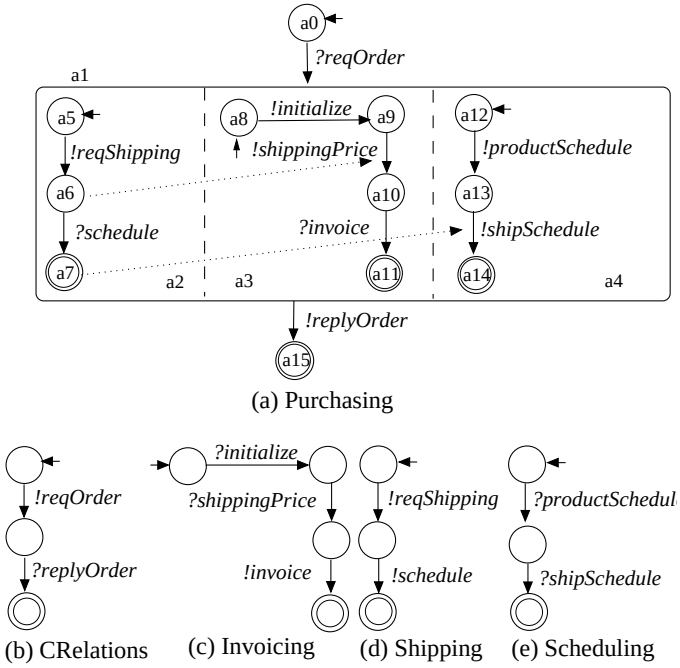


Figure 5.5: Peer interfaces for Purchase Order Handling

a_{13} to a_{14} can only be taken if the current configuration contains a_{13} and a_7.

HSMs are a variation of Statecharts [52]. The differences are, in HSMs 1) the transitions are triggered by the send or receive operations instead of events, 2) the guarding conditions of transition are defined on message contents, and 3) dependency arcs are used instead of predicates on states in the guards of the transitions. We also restrict the HSMs so that: 1) the labels of the substates of an And state are disjoint, 2) there are no transitions between the substates of an And state, and 3) an HSM can leave a state only if that state is exit-ready which means that all of its substates are final states.

The Travel Agency service is a composition of five peers, and one can specify the

interfaces of these peers with five HSMs as shown in Figure 5.4. (For readability, the guarding conditions for send transitions are not shown in the figure.) Let us consider the peer interface of the TravelAgent given in Figure 5.4(a). The $root$ state (which is not shown in the figure and is an `Or` state) has three substates: x_1, x_2 and x_{19}. The states x_1 and x_{19} are atomic states, x_1 is a default state and x_{19} is a final state. The state x_2 is an `And` state and has two substates x_3 and x_4 which are `Or` states. The substates of x_3 are x_5, x_6 and x_7. The state x_7 is an `And` state and its substates are x_8 and x_9 which are `Or` states.

HSMs provide a compact model which represents the natural hierarchy in the interface behavior. Consider the peer interface of the TravelAgent peer given in Figure 5.4(a). This interface is specified with 19 states (excluding the $root$ state) and 13 transitions. If we specify this interface with a flat finite state machine there would be 35 states and 78 transitions.

The HSM Model

Here we define the HSM model formally. An HSM is a tuple $I = (Q, \mathcal{C}, c_0, F, \delta, M, D)$ where Q is the set of states (including the $root$ state), $\mathcal{C}$ is the set of configurations, $c_0 \in \mathcal{C}$ is the initial configuration, $F \subseteq \mathcal{C}$ is the set of final configurations, δ is the transition relation, M is the set of messages, and D is the set of dependency arcs. We use $class(M)$ to denote the set of message classes and for each message $m \in M$ we use $class(m)$ to denote the message class of m. For each composite state $q \in S$, we use $sub(q)$ to denote its substates. For a substate q, $super(q)$ denotes its superstate. We use $sub^+(q)$ and $super^+(q)$ to denote all the descendants

and ancestors (respectively) of the state q in the state hierarchy. We also define the following: $sub^*(q) = sub^+(q) \cup \{q\}$ and $super^*(q) = super^+(q) \cup \{q\}$. Given a state q, $type(q) \in \{$ `Atomic`, `Or`, `And` $\}$ denotes its type.

A configuration $c \in \mathcal{C} \cup \mathcal{C}_{sub}$ is a tree whose nodes are states from Q. We denote the root node of a configuration c as $root(c)$. We use $\mathcal{C}$ to denote the configurations whose root is the *root* state, and the rest of the configurations are called subconfigurations, denoted by $\mathcal{C}_{sub}$. Given a configuration c and a state q that appears in q, ***subconf***(q, c) is the set of subtrees (subconfigurations) of c for which the root node is a substate of q, i.e., $c' \in$ ***subconf***$(q, c) \Rightarrow super(root(c')) = q$. We use ***subconf***$^+(q, c)$ to denote all the subtrees of the state q in c. All configurations have to satisfy the following constraints: 1) The root state of all the configurations in c is $root$; 2) Children of a node in a configuration are substates of that node; 3) All the leaf nodes in a configuration are `Atomic` states; 4) Each `Or` state in a configuration has exactly one child which is one of its substates; and 5) Each `And` state in a configuration has all of its substates as its children.

The transition relation δ is partitioned into send transitions δ_S and receive transitions δ_R. A send transition is of the form $(r, !z, g, u, r') \in \delta_S$ where $r \in Q$ is the source state, $r' \in Q$ is the target state, $z \in class(M)$ is the message class of the message that is being sent, u is the update function similar to the update function in peer interfaces with flat state machines, $g = g_v \wedge g_p$ is the guarding condition, g_v is the condition defined on the contents of the latest instances of the messages that have been transmitted, and g_p is the condition defined on the contents of the message that is being sent. A receive transition is of the form $(r, ?z, g, u, r') \in \delta_R$ where $r \in Q$ is

the source state, $r' \in Q$ is the target state, $z \in class(M)$ is the message class of the message that is being received, u is the update function, and g is the guarding condition that always returns TRUE. A dependency arc $d \in D$ is a state-transition pair, i.e., $D \subseteq Q \times \delta$. A transition can only be taken in a configuration which contains all the states that are the source of a dependency arc that points to that transition.

In the initial configuration c_0, all the states which are substates of an `Or` state are default states. A configuration *c* is a final configuration (i.e., $c \in F$) if all the states in c which are substates of an `Or` state are final states. We call a state q in a configuration *c* exit-ready if all the states in *subconf*(q, c) which are substates of an `Or` state are final states. A configuration *c* is a final configuration if $root(c)$ is exit-ready. A transition from a state in a configuration can only be taken if that state in that configuration is exit-ready.

The execution of an HSM starts from the initial configuration and continues by transitioning to a next configuration of the current configuration until one of the final states is reached. In Figure 5.7 we show the next configuration computation for HSMs. Given a current configuration and a message, the **takeReceiveTrans** function computes all the next configurations after receiving that message, and the **takeSendTrans** function computes all the next configurations after sending that message using the recursive functions **next** and **construct**.

5.5.1 Interactions of Hierarchical Interfaces

In this section, we discuss the semantics of a composite web service based on the peer controller pattern whose interfaces are defined with HSMs. In our model,

```
takeSendTrans(m:Message, C:set of Configurations)
returns a set of Configurations
   N: set of Configurations
   for each c in C add all elements of next(c,m,"send") to N
   store the content of m as the latest transmitted message of its type
   return N

takeReceiveTrans(m:Message, C:set of Configurations)
returns a set of Configurations
   N: set of Configurations
   for each c in C add all elements of next(c,m,"receive") to N
   store the content of m as the latest transmitted message of its type
   return N

next(c:Configuration, m:Message, direction:String)
returns a set of Configurations
   T: set of Transitions; R, N: set of Configurations
   if direction = "send" then T := findSendTrans(c,m)
   else T := findReceiveTrans(c,m)
   if T ≠ φ then
     if type(root(c)) = Atomic or c is exit-ready then
       for each t ∈ T where q' is the target of t
           find q_a s.t. q_a ∈ super*(root(c)) ∩ super*(q')
                and sub(q_a) ∉ super*(root(c)) ∩ super*(q')
           add construct(q',q_a) to R
       return R
   if T = φ and type(root((c)) = Atomic then return φ
   for each c' ∈ subconf(root(c),c)
       N := next(c',m,direction)
       for each c'' ∈ N
           make a copy of c replacing c' with c''
           add the copy to R
          else if c is exit-ready then add c'' to R
   return R
```

Figure 5.6: Next configuration computation, part 1

findSendTrans(c:Configuration, m:Message)
returns a set of Transitions
  T: set of Transitions
  for each $t = (q_1, !class(m), g_v \wedge g_p, u, q_2) \in \delta_S$ s.t. q_1 = *root*(*c*)
    if there is a dependency $(q_d, t) \in D$ s.t. q_d is not in *c* then
      continue
    if the contents of the latest transmitted messages satisfy g_v
      and the contents of m satisfy g_p then
        add t to T
  return T

findRecvTrans(c:Configuration, m:Message)
returns a set of Transitions
  T: set of Transitions
  for each $t = (q_1, ?class(m), g, u, q_2) \in \delta_R$ s.t. q_1 = *root*(*c*)
    if there is a dependency $(q_d, t) \in D$ s.t. q_d is not in *c* then
      continue
    add t to T
  return T

construct(q_t,q:State) returns a Configuration
  c: Configuration;
  set *root*(*c*) to *c*
  if *type*(q) = And then
    for each $q' \in$ *sub*(q) add **construct**(q_t,q') to *subconf*(q,*c*)
  else if *type*(q) = Or then
    if $q \in$ *super*$^+$(q_t)
      find $q' \in$ *sub*(q) s.t. $q' \in$ *super**(q_t)
    else let $q' \in$ *sub*(q) be the default state
    add **construct**(s_t,s') to *subconf*(q,*c*)
  else *subconf*(q,*c*) := ϕ
  return *c*

Figure 5.7: Next configuration computation, part 2

HSMs interact with each other by exchanging messages through FIFO queues. We assume that each HSM is associated with an input message queue and the messages are delivered without any error (i.e., no duplicate, lost or modified messages during transmission). When a message m is sent from the peer i to the peer j, the message m is inserted to the end of the input message queue of the peer j, and the configuration of the peer i is updated according to the **takeSendTrans** function given in Figure 5.7. A peer receives a message by consuming the first element of its input message queue and updates its current configuration according to the **takeReceiveTrans** function given in Figure 5.7.

Formally, a composite service with hierarchical interfaces is a tuple $HW = (class(M), I_1, \ldots, I_k)$ where $class(M)$ is a finite set of message classes, M is a finite set of messages, k is the number of participating peers, for each $1 \leq i \leq k$, $I_i = (Q_i, \mathcal{C}_i, c_{0_i}, F_i, \delta_i, M_i, D_i)$ is an HSM defining the interface of peer i, and $M = \bigcup_{1 \leq i \leq k} M_i$.

We define the execution semantics of a composite service as a transition system $T(HW) = (IT, CT, RT)$ where CT is the set of configurations, $IT \subseteq CT$ is the set of initial configurations, and RT is the transition relation of the system. The set of configurations is defined as $CT = \mathcal{C}_1 \times \Theta_1 \times \cdots \times \mathcal{C}_k \times \Theta_k$ where k represents the number of peers in the composition and Θ_i is the set of configurations of the input queue of peer i.

We introduce the following notation. Given a configuration $c \in CT$ and a peer identifier i, $c(\mathcal{C}_i)$ denotes the configuration of the peer interface I_i in configuration c, and $c(\Theta_i)$ denotes the configuration of the input queue Θ_i in configuration c.

The set of initial configurations of $T(CS)$ is defined as

$$IT = \{c \mid c \in CT \ \wedge \ (\forall 1 \leq i \leq k, c(\Theta_i) = \langle\rangle \ \wedge \ c(\mathcal{C}_i) = c_{0_i})$$

We define the following relation for a send transition which sends a message m:

$$\begin{aligned} RT_{!m} = \ & \{(c, c') \mid c, c' \in CT \wedge (\exists 1 \leq i \leq k, c'_i \in I_i.\textbf{takeSendTrans}(m, c_i) \\ & \wedge c(\mathcal{C}_i) = c_i \wedge c'(\mathcal{C}_i) = c'_i \wedge (\forall 1 \leq j \leq k, j \neq i, c'(\mathcal{C}_j) = c(\mathcal{C}_j))) \\ & \wedge \textit{receiver}(m) = I_p \ \wedge \ c'(\Theta_p) = \textit{append}(c(\Theta_p), \langle m\rangle) \\ & \wedge (\forall 1 \leq l \leq k, l \neq p, c'(\Theta_l) = c(\Theta_l))\} \end{aligned}$$

We define the following relation for a receive transition which receives a message m:

$$\begin{aligned} RT_{?m} = \ & \{(c, c') \mid c, c' \in CT \wedge (\exists 1 \leq i \leq k, c'_i \in I_i.\textbf{takeReceiveTrans}(m, c_i) \\ & \wedge c(\mathcal{C}_i) = c_i \wedge c'(\mathcal{C}_i) = c'_i \wedge (\forall 1 \leq j \leq k, j \neq i, c'(\mathcal{C}_j) = c(\mathcal{C}_j)) \\ & \wedge \textit{first}(c(\Theta_i)) = m \wedge \textit{append}(\langle m\rangle, c'(\Theta_i)) = c(\Theta_i) \\ & \wedge (\forall 1 \leq l \leq k, l \neq i, c'(\Theta_l) = c(\Theta_l)))\} \end{aligned}$$

Finally, the transition relation RT for the $T(HW)$ is

$$RT = \bigcup_{m \in M} (RT_{!m} \cup RT_{?m})$$

Having defined the execution semantics of a composite service whose peer interfaces are HSMs, we can define the conversations generated by executions of a composite service. An execution sequence $exe = c_0, c_1, ...$ is a sequence of configurations where for each $i \geq 0$, $(c_i, c_{i+1}) \in RT$ and $c_0 \in IT$. The conversation $conv(exe)$ generated by an execution sequence exe is defined recursively

as follows: The conversation $conv(c_0)$ is the empty sequence. The conversation $conv(c_0, c_1, ..., c_n, c_{n+1})$ is equal to $conv(c_0, c_1, ..., c_n), m$ if there exists a queue configuration Θ_j such that $c_{n+1}(\Theta_j) = append(c_n(\Theta_j), \langle m \rangle)$, and it is equal to $conv(c_0, c_1, ..., c_n)$ otherwise. A conversation is a complete conversation if in the last configuration of the execution sequence each peer is in a final configuration and all the message queues are empty.

5.5.2 Synchronizability Analysis for Hierarchical Interfaces

In this section we present a synchronizability analysis for composite web services whose peer interfaces are specified as HSMs. This analysis checks the two sufficient conditions for synchronizability on HSMs without flattening. Since, on average, the number of states and transitions in an HSM are less than an equivalent flat finite state machine, the analysis performed on HSMs is more efficient.

Checking Autonomous Condition

The autonomy checking algorithm is given in Figure 5.8. The base condition is that given a state, all the transitions originating from that state should be either send transitions or receive transitions. If this condition is satisfied, then we investigate whether all of the substates of that state satisfy this condition as well.

The function **simpleAutonomy** in Figure 5.8 first checks this base condition. If there are no failures, it examines the transitions whose source and target states do not share the same superstate. For each composite state q, the transition type from q should be the same as the transition type from its final substates to the states that are

```
autonomy() returns boolean
    for each q ∈ sub(root)
        status := simpleAutonomy(q)
        if status = "fail" return false
        if q is final return finalStateCheck(q)
    return true

simpleAutonomy(q:State) returns String
    mystatus:String
    mystatus := "none"
    if there is a send transition from q then
        mystatus := "send"
    if there is a receive transition from q then
        if mystatus = "send" then return "fail"
        else mystatus := "receive"
    for each q' ∈ sub(q)
        status := simpleAutonomy(q')
        if status = "fail" then return "fail"
        if there exists a transition t from q'
          s.t. target of t is not in sub+(q) then
          if status = "send" and mystatus = "receive" then return "fail"
          if status = "receive" and mystatus = "send" then return "fail"
          if status ≠ "none" then mystatus := status
    return mystatus

finalStateCheck(q:State) returns boolean
    if there exists a transition from q then return false
    for each q' ∈ sub(q)
        if q' is final and finalStateCheck(q') = false then return false
    return true
```

Figure 5.8: Autonomy check

not the substates of q. For example, adding a receive transition with source state a_{14} and target state a_{15} to the peer interface in Figure 5.5(a) would violate the autonomy. On the other hand, adding a send transition with the same source and target states does not violate the autonomous condition.

```
synchronous(Peers: set of HSMs) returns boolean
    inspect:set of Configuration arrays, a, b:Configuration array
    sendConf, recvConf:set of Configurations, legal:boolean
    inspect := φ
    for each I_i ∈ Peers a[i] := I_i.c_0
    add a to inspect
    while inspect ≠ φ do
        remove one element from inspect and write it to a
        for each Message m ∈ M
            for each I_i ∈ Peers
                legal := false
                sendConf := I_i.takeSendTrans(m, {a[i]})
                if sendConf≠ φ then
                  for each I_j ∈ Peers s.t. I_j ≠ I_i
                      recvConf := I_j.takeReceiveTrans(m,{a[j]})
                      if recvConf ≠ φ then
                        legal=true
                        for each sc ∈ sendConf
                            for each rc ∈ recvConf
                                let b be a copy of a
                                set b[i] to sc and b[j] to rc
                                add b to inspect
                  if legal=false then return false
    return true
```

Figure 5.9: Synchronous compatibility check

The autonomy check algorithm first invokes the **simpleAutonomy** for all substates of the $root$. If no failures are reported, the algorithm checks that there are no configurations in $\mathcal{C}$ that follow the configurations in the final configuration set (F) of the HSM in question. If no violations are found, the algorithm concludes that the HSM is autonomous.

Checking Synchronous Compatibility Condition

To check the synchronous compatibility of the peers in a composite web service, we search for the illegal configurations in the Cartesian product of the peer interfaces. The configurations of this product are tuples with the domain $\mathcal{C}_1 \times \cdots \times \mathcal{C}_k$ where k is the number of peers in the composition and $\mathcal{C}_i$ is the configuration set of peer i for $1 \leq i \leq k$. Let c_i denote the configuration for peer interface I_i at a product configuration. A configuration in the product is illegal if I_i.**takeSendTrans**(m, c_i) returns a nonempty set while for all other peers I_j.**takeReceiveTrans**(m, c_j) is empty. The synchronous compatibility checking algorithm is given in Figure 5.9.

5.5.3 BPEL Generation from Hierarchical Interfaces

We have implemented a translator that takes HSMs defining the peer interfaces and automatically creates a BPEL specification for each participant peer to publish. The translator first synthesizes WSDL specifications, which are the connectivity contracts, and then generates the BPEL specifications. A generated BPEL specification contains partner link definitions to access other peers, variable declarations, and behavior description of the peer.

To reflect the hierarchy of the peer interface, the behavior description in a generated BPEL specification consists of nested scopes. Each scope has two local variables. The variable `state` represents current state in the scope, and the variable `exit` is used for exiting the scope. Each scope is a `while` loop that terminates depending on the value of `exit`. The body of the loop is a `switch` activity whose

`case` conditions are defined on the value of the `state`.

A `case` block implementing the operations at the corresponding state, consists of two activities. The type of the first activity depends on the type of the state. If the state is of type `And`, a `flow` is generated that has one subactivity per its substate. If the state is of type `Or`, a new scope is generated and this inner scope's local `state` is set to the default substate. The second activity corresponds to the transitions originating from this state. The mappings of send and receive transitions to BPEL code fragments are similar to the mapping discussed in Section 5.4. In this second activity, if the state is an exit-state then the local `exit` variable is set.

The following is an excerpt from the BPEL specification generated from the implementation of the peer interface in Figure 5.4(a). This excerpt shows the code fragment synthesized for state x_2.

```
<case condition="state='x2'">
  <sequence>
   <flow>
     <scope>
       <variables> ...</variables>
       <!-- define fresh state and exit variable-->
       <sequence>
         ...<!--initialize state and exit variables-->
         <while condition="exit!='yes'">
           <switch>
             <case condition="state='x7'">
              <sequence>
               <flow>
                ...<!--car reservation-->
                ...<!--flight reservation-->
               </flow>
               <assign><copy>
                 <from expression="'yes'"/> <to variable="exit"/>
               </copy></assign>
              </sequence>
             </case>
```

```
            ...<!--cases when state is 'x5' or 'x6'-->
          </switch>
        </while>
      </sequence>
    </scope>
    <scope>
      ...<!--hotel reservation -->
    </scope>
  </flow>
    ...<!--send travelInv, and assign 'x19' to state-->
 </sequence>
</case>
```

If there are transitions whose source and target states are not siblings, the translator identifies these transitions during preprocessing and flattens that portion of the HSM.

When there are dependency arcs in a peer interface, the translator generates one `link` [20] per arc. The translator places the link target and link source inside the corresponding send or receive fragments. Consider the dependency arc in Purchasing peer (Figure 5.5(a)) which implies that to take the transition from a_9 to a_{10} the current configuration has to contain the state a_6. For this dependency arc, the translator puts the link source inside the activity that sets the state to a_6, and puts the link target into the fragment corresponding to the transition from a_9 to a_{10}.

5.6 Experiments and Verification Results

In this section, we present the experimental results on all of the composite web services introduced in Section 5.1 that are implemented based on the peer controller pattern. We implemented the peer interfaces of the Loan Approval and the BookCD

Order services by using the `StateMachine` class, which is flat finite state machine implementation. The peer interfaces of the Travel Agency and the Purchase Order Handling services are implemented by using the `HSM` class, i.e., the peer interfaces at these composite web services are specified with HSMs. We verified these implementations using the modular verification approach presented in this chapter. In our approach, by exploiting the structure of the peer controller pattern, verification of the peer implementations with respect to their interfaces is performed separately from the verification of the conversations (i.e., the global behavior) of the composite web service. As we demonstrate below, this is crucial for feasibility of the automated verification of composite web services.

Next, we first discuss the behavior verification results in our experiments with these examples. Then, we continue with interface verification discussion.

Based on our modular verification approach, we verified the global behavior of the Loan Approval service with the SPIN model checker using the conversation model. An example property we verified during behavior verification is the following: "Whenever a *request* message with a small amount is sent, eventually an approval message accepting the loan request will be sent." The behavior verification took less than one second and used 1.49MB memory. During the behavior verification, we observed that the reachable state space of the Loan Approval system is finite (154 states). Independent of the size of the message queues, during any execution, there is at most one message in each queue at any state; therefore, increasing the size of message queues did not increase the state space. Note that, this experimental observation is not a proof that the results we obtained using bounded verification will

hold for the Loan Approval service when unbounded message queues are used. To guarantee this, we used synchronizability analysis. Our automated synchronizability analyzer identified Loan Approval service as synchronizable. Therefore, we were able to verify Loan Approval service using synchronous communication, and since it is synchronizable, the verification results are guaranteed to hold when unbounded message queues are used.

On the other hand, during the execution of the BookCD Order service, the number of messages in the message queues is not bounded. We have verified the behavior of this example with different queue sizes. As shown in Figure 5.10, the state space increases exponentially with the size of the queues. In fact, the number of reachable states for this example is infinite if unbounded queues are used. The exponential growth in the state space affects the performance of SPIN significantly. SPIN ran out of memory when the queue size was set to 15. On the other hand, our automated synchronizability analyzer has identified this example as synchronizable. Therefore, we can verify this composite service by replacing asynchronous communication with synchronous communication without changing the conversation set generated by this composite web service. With synchronous communication there are only 68 states and the behavior verification have succeeded in a fraction of a second and used 1.49 MB memory. Since with synchronous communication the messages are not buffered, the state space of the specification is reduced and this reduction leads to a significant improvement in the efficiency of the behavior verification and avoids the state space explosion in this example.

Another composite web service with infinite state space is the Travel Agency

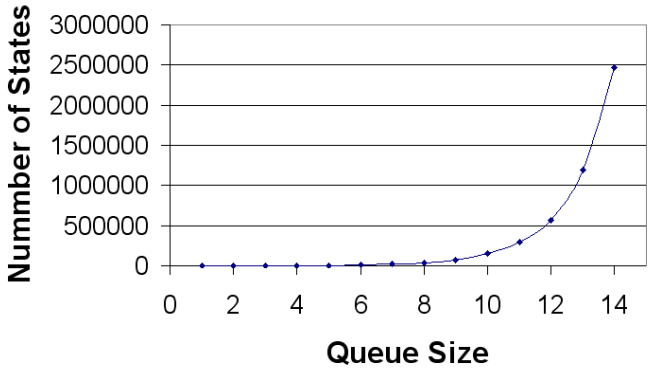


Figure 5.10: Effect of the queue sizes on the state space

service. During behavior verification of this example with asynchronous communication, we observed that for the Travel Agency service the state space increases exponentially with the size of the queues. This exponential growth affected the performance of the SPIN model checker and it ran out of memory for queue size 12. We applied the synchronizability analysis to the Travel Agency example, and identified that it is synchronizable. Therefore, we were able to verify the Travel Agency service with synchronous communication without affecting its conversation set. With the synchronous communication state space contains only 8911 states, and the behavior verification with SPIN succeeded in 0.38 seconds and used 5.15 MB memory.

The behavior verification results for the Purchase Order Handling service show that the synchronizability analysis is not only useful for unbounded behavior verification but also for improving the efficiency of the verification performance. We observed that the Purchase Order Handling service has finite set of reachable states

since there is at most one message at each queue at any state during any execution. The behavior verification with asynchronous communication used 6.78 MB memory in 0.59 seconds for different queue sizes. During these experiments with asynchronous communication, increasing the size of message queues did not increase the state space which is 10105 states. We applied the synchronizability analysis and identified that the five HSMs given in Figure 5.5, are synchronizable. With synchronous communication there are only 1562 states, and behavior verification with SPIN took 0.08 seconds and used 2.01 MB memory.

After the behavior verification, we performed the interface verification on these examples. The interface verification performance is displayed in Table 5.1. The first part shows the JPF performance for the examples whose peer interfaces are specified as flat state machines (Loan Approval and BookCD Order), the second part shows the performance for the examples whose peer interfaces are specified as HSMs (Travel Agency and Purchase Order Handling). In this table, the interface verification for TravelAgent and Purchasing peers are given when the corresponding peer is implemented as one thread. One can use multiple threads for implementing the TravelAgent and Purchasing peer to exploit the concurrency introduced by the And state in the interface specification. For such an implementation, we need to verify that the combined behavior of the concurrent threads conform to the peer interface. One can use the modular verification approach presented in Chapter 4 and verify the concurrent threads separately.

Using peer interfaces as communication stubs and abstracting away the non-control attributes of messages with message stubs have lead to a significant im-

Table 5.1: Interface verification performance

Loan Approval Service			BookCD Order Service		
Peer	Time(s)	Memory(MB)	Peer	Time(s)	Memory(MB)
CustomerRelations	8.86	3.84	Client	3.64	4.84
LoanApprover	9.65	4.70	Supplier	3.43	4.63
RiskAssessor	8.15	3.64			
Travel Agency Service			**Purchase Order Handling Service**		
Peer	Time(s)	Memory(MB)	Peer	Time(s)	Memory(MB)
Customer	5.63	4.92	CRelations	4.63	3.73
HotelReserve	4.76	3.95	Invoicing	4.54	4.22
CarReserve	4.61	3.74	Shipping	4.81	3.91
FlightReserve	4.83	3.89	Purchasing	5.00	7.69
TravelAgent	9.72	19.69	Scheduling	4.83	3.76

provement in the efficiency of the interface verification. Also isolating the peers with simple drivers described above and verifying only one session improved the efficiency further. Therefore, in both examples JPF spent less than 10 seconds and used less than 5 MB memory.

We also tried to verify the whole Loan Approval service using JPF without separating the interface and behavior verification steps. The first problem is that JPF cannot handle asynchronous communication among peers. To overcome this problem, we wrote some Java code that simulates the JAXM provider and the asynchronous input queues. In this simulation, for each peer (aside from the application thread) there is a concurrent queue instance and a thread which is activated whenever a

message arrives in the queue. We ran JPF on this simulation program for only one session. JPF ran out of memory without producing a conclusive result. Hence, without using the modular approach proposed in this chapter, JPF is unable to verify properties of the Loan Approval service.

Our experiments show that the modularity in the verification process based on the peer controller pattern improves the efficiency of verification of composite web services significantly. We can verify asynchronously communicating web service implementations using reasonable amount of time and memory which are otherwise too large for a Java model checker to handle. With the aid of synchronizability analysis, during behavior verification, we can reason about the global behavior with respect to unbounded queues and perform the behavior verification efficiently. Furthermore, the usage of stubs during the interface verification causes a significant reduction in the state space, thus improving the performance of the verification process.

5.7 Related Work

Recently, some researchers have used existing model checking tools to assure reliability of web services. Nakajima [79] and Fu et al. [47] verify a given web service flow (specified in WSFL and BPEL respectively) by using the explicit state model checker SPIN [58]. Foster et al. [56] use the Labeled Transition System Analyzer (LTSA) for inferring the correctness of the web service compositions which are specified using message sequence charts. Narayanan et al. [80] verify web services

using a Petri Net model generated from a DAML-S description of a service. An extended Petri Net model is proposed by Verbeek [98] for verification of workflows and analyzing inheritance relations among models. These earlier verification efforts focus on the specification level and do not address the correctness of individual peer implementations. Verification of the communication flow does not guarantee that a composite web service behaves according to its specification unless we can ensure that each peer obeys its published contract (this requirement is called *conformance* by Meredith et al. [76]). In the DFV approach presented in this chapter, both interaction behavior and interface conformance are verified.

Statecharts, which are a general form of our HSM model, have been used to describe web service behavior [8, 73, 70]. Their aim is to declare service compositions. Unlike these studies, we use HSMs not only for specifying web services but also for verifying their global interaction properties.

Chapter 2 has discussed related work regarding to the peer controller pattern. The verification aspects of these works are also related to the DFV approach presented here.

Chapter 6

Conclusion

This dissertation presents a Design for Verification (DFV) approach for concurrent programming to eliminate synchronization errors in concurrent Java programs and for composite web services to automatically analyze interaction properties between the participating peers. The approach is realized via two design patterns that decouple behavior and interface specifications. Based on this decoupling we have developed a modular assume-guarantee style verification strategy. We use Java PathFinder [99] for interface verification in both application domains. We use the Action Language Verifier [23] for behavior verification of concurrency controllers and, we use SPIN [58] for behavior verification of composite web services. Modularization of the verification task improves the scalability of the verification and helps us combine different approaches to verification with their associated strengths.

Our experiments with two-real life concurrent applications demonstrated the effectiveness, the applicability, and the scalability of our DFV for concurrent programming. The experimental study on TSAFE with fault seeding resulted in a classification of faults that our technique can identify. During these case studies the inter-

face verification was the most challenging part whereas domain specific behavior verification was more efficient. Due to behavior encapsulation of the concurrency controller pattern, the extracted behavior model was compact and suitable for automated verification. The interface verification was the bottleneck even though it is performed on isolated threads. On the other hand, we have achieved a significant improvement in interface verification because of the thread isolation and usage of controller interfaces as stubs. Without these techniques a model checker like Java Pathfinder cannot handle these examples at all. On the whole, the presented verification technique was able to find almost all of the seeded faults. The isolation and substitution of controller interfaces has a dramatic impact on the efficiency of the interface verification.

In the second application domain, which is the composite web services, we can reason about the global behavior with respect to unbounded queues and improve the efficiency of the behavior verification because of the adaption of the synchronizability analysis proposed in [47]. Also, using the peer interfaces in place of the asynchronous communication component abstracts the asynchronous messaging leading to a significant improvement in the efficiency of the interface verification. In this domain, another benefit of explicit peer interfaces is that they can be used to improve interoperability. To support this, we automatically generate BPEL specifications from the peer interfaces. We extend the peer interfaces to HSMs to reflect the natural hierarchy of the peer behavior. Our experiments with several composite web service instances show that the DFV approach for asynchronously communicating web services enables the verification of the web service implementations as well as

the properties of interactions among the participant peers using reasonable amount of time and memory.

The presented DFV approach is a significant step toward the scalability of automated software verification. Our experiments on both concurrent programs and web services show that automated software model checking can become feasible for real-life applications when the presented DFV principles are applied.

Currently, we use ACTL properties in the DFV approach for concurrent programming. Although the model checker we use can perform CTL verification, in the current formalism we have showed that the ACTL properties of controller behavior are preserved in the program if all the threads are interface correct. In the future, with some assumptions on the scheduler, we may be able to use full CTL properties in the DFV approach for concurrent programming.

Another future direction is automated discovery of forgotten shared objects. We would like to extend our verification framework with an escape analysis technique to handle such situations. Escape analysis techniques are used to identify the objects which escape from a scope (thread or method). The publicly available escape analysis tools we experimented so far [19, 60] either do not scale to programs as big as TSAFE or identify too many objects as shared. In the future we plan to develop a specialized escape analysis to identify the objects which need to be synchronized without considering one-time-written shared objects.

In the long run, we would like to investigate ways to apply the DFV principles to other aspects of software systems. In this direction we would like to extend the DFV approach to other programming paradigms such as sequential programming

and aspect oriented programming with new verifiable design patterns.

Bibliography

[1] J. Adams, S. Koushik, G. Vasudeva, and G. Galambos. *Patterns for e-business: A Strategy for Reuse*. IBM Press, 2001.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 98–109, 2005.

[3] T. Austin. Design for verification? *IEEE Design & Test of Computers*, 18(4):77–80, 2001.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN Workshop on Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, 2001.

[5] T. Ball and S. K. Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN - SIGACT*

Symposium on Principles of Programming Languages (POPL), pages 1–3, Portland, USA, January 2002.

[6] R. Behrends and R. E. K. Strirewalt. The universe model: An approach for improving the modularity and reliability of concurrent programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'2000)*, pages 20–29, 2000.

[7] B. Benatallah, M. Dumas, M. Fauvet, F. Rabhi, and Q. Z. Sheng. Overview of some patterns for architecting and managing composite web services. *ACM SIGecom Exchanges*, 3(3):9–18, August 2002.

[8] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. H. H. Ngu. Declarative composition and peer-to-peer provisioning of dynamic web services. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 297–308, 2002.

[9] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proceedings of the 1st International Conference on Service Oriented Computing*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer, 2003.

[10] D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Synthesis of composite e-services based on automated reasoning. In *Pro-*

ceedings of the 2nd International Conference on Service Oriented Computing, pages 105–114, 2004.

[11] A. Betin-Can and T. Bultan. Interface-based specification and verification of concurrency controllers. In *Proceedings of the Workshop on Software Model Checking(SoftMC), Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 89, 2003.

[12] A. Betin-Can and T. Bultan. Interface-based specification and verification of concurrency controllers. Technical Report 2003-13, Computer Science Department, University of California, Santa Barbara, June 2003.

[13] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 248–257, 2004.

[14] A. Betin-Can and T. Bultan. Verifiable web services with hierarchical interfaces. In *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005)*, pages 85–94, 2005.

[15] A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In *Proceedings of the 14th International World Wide Web Conference*, pages 750–759, Chiba, Japan, May 2005.

[16] A. Betin-Can, T. Bultan, M. Lindvall, S. Topp, and B. Lux. Application of design for verification with concurrency controllers to air traffic control

software. In *Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE)*, 2005.

[17] D. Beyer, A. Chakrabarti, and T. A. Henzinger. An interface formalism for web services. In *Proceedings of the 1st International Workshop on Foundations of Interface Technologies, to appear*, ENTCS. Elsevier, 2005.

[18] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *Proceedings of the 14th International World Wide Web Conference*, pages 148–159, Chiba, Japan, May 2005. ACM Press.

[19] J. G. Bogda. *Program Analysis Alleviates Java Synchronization*. PhD thesis, University of California, Santa Barbara, 2001.

[20] Business process execution language for web services version 1.1. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, May 2003.

[21] T. Bultan and A. Betin-Can. Scalable software model checking using design for verification. In *Proceedings of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments*, pages 10–14, Zurich, Switzerland, October 2005.

[22] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proceedings of the Twelfth International World Wide Web Conference (WWW 2003)*, pages 403–410, Budapest, Hungary, May 2003.

[23] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 382–386, 2001.

[24] T. Cargill. Specific notification for Java thread synchronization. In *Proceedings of the 3rd Conference on Pattern Languages of Programs*, 1996.

[25] J. C.Corbett, M. B.Dwyer, J. Hatcliff, S. Laubach, C. S. Pasarenau, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 439–448, 2000.

[26] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering*, pages 385–395, 2003.

[27] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdziński, and F. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 2404, pages 428–441. Springer-Verlag, 2002.

[28] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 34, October 1999.

[29] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[30] W. Crawford and J. Kaplan. *J2EE Design Patterns*. O'Reilly and Associates Inc., Sebastopol,California, 2003.

[31] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120, 2001.

[32] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[33] R. DeLine and M. Fahndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming ECOOP*, pages 465–490, 2004.

[34] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68, 2000.

[35] X. Deng, M. B. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering*, pages 442–452, 2002.

[36] G. Dennis. TSAFE: Building a trusted computing base for air traffic control software, Master's Thesis, 2003.

[37] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, 2001.

[38] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser. Analyzing interaction orderings with model checking. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 154–163, 2004.

[39] H. Erzberger. The automated airspace concept. In *Proceedings of the 4th USA/Europe Air Traffic Management R&D Seminar*, 2001.

[40] H. Erzberger. Transforming the NAS: The next generation air traffic control system. In *Proceedings of the 24th International Congress of the Aeronautical Sciences*, 2004.

[41] Advance automation system. Department of Transportation, Office of Inspector General, Audit Report, AV-1998-113, 1998.

[42] M.-C. Fauvet, M. Dumas, F. Rabhi, and B. Benatallah. Patterns for e-services composition. In *In Proceedings of the 3rd Asia-Pacific Conference on Pattern Languages of Programs*, 2002.

[43] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 29th ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 234–245, 2002.

[44] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 213–224, 2003.

[45] X. Fu. *Formal Specification and Verification of Asynchronously Communicating Web Services*. PhD thesis, University of California, Santa Barbara, 2004.

[46] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA)*, Lecture Notes in Computer Science, pages 188–200, 2003.

[47] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th International World Wide Web Conference*, pages 621–630, 2004.

[48] C. E. Gerede, R. Hull, O. H. Ibarra, and J. Su. Automated composition of e-services: Lookaheads. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 252–262, 2004.

[49] P. Godefroid. Model checking for programming languages using VeriSoft.

In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186, January 1997.

[50] P. Godefroid, C. Colby, and L. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–357, June 1998.

[51] J. E. Hanson, P. Nandi, and S. Kumaran. Conversation support for business process integration. In *Proceedings of the 6th International Enterprise Distributed Object Computing Conference*, pages 65–74, 2002.

[52] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[53] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space craft controller using spin. *IEEE Transactions on Software Engineering*, 27(8):749–765, August 2001.

[54] Nasa high dependability computing program experience and knowledge base. http://www.cebase.org/HDCP/.

[55] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, Lecture Notes in Computer Science 2648, pages 235–239. Springer-Verlag, 2003.

[56] H.Foster, S. Uchitel, J.Magee, and J.Kramer. Model-based verification of web

service compositions. In *Proceedings of the 18th International Conference on Automated Software Engineering (ASE)*, pages 152–163, 2003.

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

[58] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[59] Conversation support for agents, e-business, and component integration. http://www.research.ibm.com/convsupport.

[60] Indus. http://indus.projects.cis.ksu.edu.

[61] Java 2 platform standard edition (j2se) 5.0. http://java.sun.com/j2se/1.5.0.

[62] Java 5.0 concurrency utilities. http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/.

[63] Java API for XML messaging (JAXM). http://java.sun.com/xml/jaxm/.

[64] Java Message Service. http://java.sun.com/products/jms/.

[65] Jsr-166 concurrency utilities. http://www.jcp.org/en/jsr/detail?id=166.

[66] A. Krasniewski. Design for verification testability. In *Proceedings of the conference on European design automation, EDAC*, pages 644–648, March 1990.

[67] R. S. Lazić. *A Semantic Study of Data independence with applications to model checking*. PhD thesis, Oxford University, 1999.

[68] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts, 1999.

[69] M. Lindvall, I. Rus, F. Shull, M. V. Zelkowitz, P. Donzelli, A. Memon, V. R. Basili, P. Costa, R. T. Tvedt, L. Hochstein, S. Asgari, C. Ackermann, and D. Pech. An evolutionary testbed for software technology evaluation (to appear). *NASA Journal of Innovations in Systems and Software Engineering*, 1(1):3–11, 2005.

[70] Z. Maamar, B. Benatallah, and W. Mansoor. Service chart diagrams - description & application. In *Proceedings of the 12th International World Wide Web Conference, Alternate paper tracks*, 2003.

[71] J. Magee and J. Kramer. *Concurrency: State Model and Java Programs*. Wiley, 1999.

[72] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.

[73] M. Mecella, B. Pernici, and P. Craca. Compatibility of e-services in a cooperative multi-platform environment. In *Proceedings of the 2nd International Workshop on Technologies for E-Services (VLDB-TES)*, pages 44–57, 2001.

[74] P. Mehlitz. Design for verification with dynamic assertions. Technical report, NASA Ames, July 2003.

[75] P. Mehlitz and J. Penix. Design for verification using design patterns to build

reliable systems. In *Proceedings of 6th Workshop on Component-Based Software Engineering*, 2003.

[76] L. Meredith and S. Bjorg. Contracts and types. *Communications of ACM*, 46(10):41–47, October 2003.

[77] Microsoft Message Queuing Service. http://www.microsoft.com/msmq.

[78] Message-driven thread api for the java programming language. http://mdthread.org/.

[79] S. Nakajima. Verification of web service flows with model-checking techniques. In *International Symposium on Cyber Worlds: Theories and Practice*, pages 378–385, 2002.

[80] S. Narayanan and S. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International World Wide Web Conference*, pages 77–88, 2002.

[81] O'Reilly. *Java Distributed Computing*. O'Reilly and Associates Inc., Sebastopol, California, 1998.

[82] O'Reilly. *Java Swing*. O'Reilly and Associates Inc., Sebastopol,California, 1998.

[83] J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *ACM Software Engineering Notes*, 9:177–184, 1984.

[84] OWL-S: Semantic markup for web services. http://www.daml.org/services/owl-s/1.0/owl-s.html, 2003.

[85] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 168–183, 1999.

[86] S. K. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 166–179, 2002.

[87] Java remote method invocation specification. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmi-arch3.html.

[88] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley and Sons, 2000.

[89] F. Sforza, L. Battu, M. Brunelli, A. Castelnuovo, and M. Magnaghi. A design for verification methodology. In *International Symposium on Quality Electronic Design (ISQED '01)*, pages 50–55, 2001.

[90] N. Sharygina, J. C. Browne, and R. P. Kurshan. A formal object-oriented analysis for software reliability: Design for verification. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 318–332, 2001.

[91] A. R. Silva, J. Pereira, and J. A. Marques. Object synchronizer: A design pattern for object synchronization. In *Proceedings of European Conference of Pattern Languages*, 1996.

[92] K. Sivashanmugam, K. Verma, A. P. Sheth, and J. A. Miller. Adding semantics to web services standard. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2003)*, pages 395–401, 2003.

[93] Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[94] S. D. Stoller and Y. A. Liu. Transformations for model checking distributed java programs. In *Proceedings of SPIN Workshop on Model Checking Software*, pages 192–199, 2001.

[95] O. Tkachuk and M. B. Dwyer. Adapting side-effects analysis for modular program model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 116–129, 2003.

[96] O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *In the Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, ESEC/FSE 2003*, pages 188–197, 2003.

[97] D. Trowbridge, D. Mancini, D. Quick, G. Hohpe, J. Newkirk, and D. Lavigne. *Enterprise Solution Patterns Using Microsoft .NET: Version 2.0 Patterns and Practices*. Microsoft Press, 2003.

[98] H. M. W. Verbeek. *Verification of WF-nets*. PhD thesis, Technische Universiteit Eindhoven, Netherlands, 2004.

[99] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, April 2003.

[100] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 218–228, 2002.

[101] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 184–193, 1986.

[102] Web service choreography interface 1.0. http://www.w3.org/TR/wsci/, August 2002.

[103] Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[104] Extensible markup language (XML). http://www.w3.org/XML/.

[105] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–179, 2002.