Eliminating Synchronization Faults in Air Traffic Control Software via Design for Verification with Concurrency Controllers *

Aysu Betin Can (aysu@ii.metu.edu.tr)
Informatics Institute, Middle East Technical University, 06531, Ankara, Turkey

Tevfik Bultan (bultan@cs.ucsb.edu)
Computer Science Department University of California Santa Barbara,CA 93106, USA

Mikael Lindvall (mikli@fc-md.umd.edu), Benjamin Lux (blux@fc-md.umd.edu) and Stefan Topp (stopp@fc-md.umd.edu)
Fraunhofer Center for Experimental Software Engineering, College Park, MD 20742, USA

Abstract.

The increasing level of automation in critical infrastructures requires development of effective ways for finding faults in safety critical software components. Synchronization in concurrent components is especially prone to errors and, due to difficulty of exploring all thread interleavings, it is difficult to find synchronization faults. In this paper we present an experimental study demonstrating the effectiveness of model checking techniques in finding synchronization faults in safety critical software when they are combined with a *design for verification* approach. We based our experiments on an automated air traffic control software component called the Tactical Separation Assisted Flight Environment (TSAFE). We first reengineered TSAFE using the *concurrency controller design pattern*. The concurrency controller design pattern enables a modular verification strategy by decoupling the behaviors of the concurrency controllers from the behaviors of the threads that use them using interfaces specified as finite state machines. The behavior of a concurrency controller is verified with respect to arbitrary numbers of threads using the infinite state model checking techniques implemented in the Action Language Verifier (ALV). The threads which use the controller classes are checked for interface violations using the finite state model checking techniques implemented in the Java Path Finder (JPF). We present techniques for thread isolation which enables us to analyze each thread in the program separately during interface verification. We conducted two sets of experiments using these verification techniques. First, we created 40 faulty versions of TSAFE using manual fault seeding. During this exercise we also developed a classification of faults that can be found using the presented design for verification approach. Next, we generated another 100 faulty versions of TSAFE using randomly seeded faults that were created automatically based on this fault classification. We used both infinite and finite state verification techniques for finding the seeded faults. The results of our experiments demonstrate the effectiveness of the presented design for verification approach in eliminating synchronization faults.

Keywords: model checking, concurrent programming, synchronization, design patterns, interfaces

* This work is supported by the NSF grants CCF-0341365, CCF-0614002, and CCF-0438933, the NASA funded High Dependability Computing Project (http://hdcp.org/) through NASA cooperative agreement NCC2-1968, and the TUBITAK grant 106E032.

1. Introduction

The use of software in critical infrastructures has been steadily increasing. Automation is a necessity for most critical infrastructures in order to address the demands to improve their capacity and efficiency. The ability to develop highly dependable software systems is one of the most important roadblocks in increasing the automation level in critical infrastructures. For example, the failure of the power grid in northeastern U.S. on August 14th, 2003 that caused the largest blackout in U.S. history was partly due to a software fault (Neumann, 2004). This software fault caused the alarm system at the control center of an Ohio utility company to fail, delaying the response by the operators that could have prevented the spread of the blackout. For other critical infrastructures such as the air traffic control system, software faults can have even more drastic and tragic consequences. It is necessary to develop techniques and tools that can be used to eliminate faults in software systems before they are deployed as part of such critical infrastructures.

In this paper we discuss techniques for detection and elimination of synchronization faults in safety-critical systems using a *design for verification* approach. In particular, we focus on a design for verification approach for concurrent programming in Java with the goal of eliminating synchronization faults from Java programs using model checking techniques. Concurrency is commonly used in safety critical systems, and synchronization of concurrently executing threads is difficult to get right.

Concurrent programming in Java is especially error-prone. Misuse of Java synchronization statements `synchronized`, `wait`, `notify`, and `notifyAll` can lead to common errors such as nested monitor lockouts, missed or forgotten notifications, slipped conditions, etc. (Lea, 1999). Such errors can cause a software system to deadlock or may lead to loss of integrity of shared data due to unsynchronized updates. One defensive programming approach is to declare all methods of all shared objects to be `synchronized` in order to protect the integrity of the shared data. However, excessive use of synchronization can degrade performance, and more importantly, can cause circular dependencies among the threads that are holding and requesting locks, and, therefore, lead to deadlocks.

In addition to such problems with lock acquire and release operations, waiting and notification mechanisms in Java can also create problems. Many concurrent Java programs use synchronization mechanisms that require conditional waits. A conditional wait in Java can be implemented by creating an object that corresponds to a condition, and then using that object's wait queue for waiting if the condition is false. In this approach, it is necessary to notify the thread(s) that are waiting for a condition after the execution of any action that might change that condition. In order to implement notifications in a program correctly, a programmer has to understand the dependencies

between all actions and all conditions that might depend on those actions, and notify the corresponding threads accordingly. A forgotten notification may cause a thread to get stuck at a wait statement and halt its progress.

The design for verification approach investigated in this paper is based on the concurrency controller design pattern proposed in (Betin-Can and Bultan, 2004). We believe that, for safety critical systems, the benefits of obtaining a highly dependable software system outweigh the extra effort that is required to learn and use this design for verification approach. Concurrency controller classes developed based on the concurrency controller pattern specify synchronization policies for coordinating the interactions among multiple threads. In the concurrency controller design pattern, software developers do not use the error-prone synchronization statements in Java. Synchronization statements are either provided in pre-defined classes, or generated automatically. The behavior of a concurrency controller is specified as a set of actions (forming the methods of the controller class) where each action consists of a set of guarded commands. The controller interface is specified as a finite state machine which defines the order that the actions of the controller can be executed by each thread. The responsibility of the software developer is to write the guarded commands that specify the synchronization policy and the finite state machine that specifies how the threads should use this synchronization policy.

The design for verification approach based on concurrency controllers enables efficient model checking of Java programs for finding synchronization errors. Model checking is an automated verification technique that has been successful in finding synchronization errors in concurrent systems. One difficulty in applying model checking to software verification is the issue of scalability. The exponential increase in the state-space of a software system due to increasing number of variables and concurrent components leads to the state-space explosion problem, making it impossible to exhaustively search all behaviors of the system. In order to achieve scalable software model checking, it is necessary to find ways to extract compact models from programs that hide the details that are not relevant to the properties that are being verified. The model extraction problem is typically handled using a reverse engineering step in which static analysis tools are used to rediscover information about programs that may be known to software developers at design time. A design for verification approach, that enables software developers to document the design decisions that can be useful for verification, can improve the scalability and therefore the applicability of model checking techniques significantly (Bultan and Betin-Can, 2005).

We use a modular verification strategy based on the concurrency controller pattern. During behavior verification we verify automatically generated infinite state models of concurrency controllers using the Action Language Verifier (ALV) (Bultan and Yavuz-Kahveci, 2001; Yavuz-Kahveci et al., 2005)

assuming that the threads that use the controllers obey their interfaces. During interface verification we verify this assumption using the explicit and finite state model checker Java PathFinder (JPF) (Visser et al., 2003). In this modular verification strategy, the behavior and the interface verification steps are completely decoupled. Moreover, during interface verification there is no need to consider interleavings of different threads. For each thread, we are only interested in the order of calls by that thread to the methods of its private interface controller object, and the only interaction among different threads is through the shared objects that are protected using the concurrency controllers. Based on these observations, we are able to verify each thread in isolation during interface verification. In this paper, we present techniques for isolating threads in programs with GUI components, RMI connections and network communication. We discuss generic environment models for isolating both implicitly and explicitly created threads in such programs. During thread isolation, we use a dependency analysis to identify the input parameters that influence the synchronization behavior.

We conducted an experimental study with the goal of investigating the effectiveness of the presented design for verification approach on a safety critical air traffic control software. Our experimental study is based on a software component called the Tactical Separation Assisted Flight Environment (TSAFE). TSAFE is part of a framework developed by NASA researchers which targets the automation of the air traffic control system. TSAFE is an implementation of a safety critical component in this framework.

In order to use it in our experiments we first reengineered TSAFE based on the concurrency controller design pattern. Then, we created 40 new versions of the TSAFE source code by manual fault seeding. The 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 a shared object. We also created another 100 new versions of the TSAFE source code by automatically seeding randomly generated faults. 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. It also resulted in improvements to our verification techniques by helping us focus on their weaknesses observed during the experiments. During this experimental study we also developed a classification of the faults that can be found using our framework.

The rest of the paper is organized as follows. In Section 2 we briefly describe the TSAFE software. In Section 3 we give an overview of the concurrency controller pattern and explain how we reengineered the TSAFE software based on this pattern. In Section 4 we discuss the behavior verification for the concurrency controllers used in TSAFE, and in Section 5 we

discuss the interface verification for the client and server components. We also present the techniques we used to isolate the threads during interface verification in Section 5. We present and discuss the results of our experiments in Section 6. After the discussion of the related work in Section 7, we state our conclusions in Section 8.

2. Automated Airspace Concept and TSAFE

The main goal of the air traffic control system is to keep a safe distance between the aircraft while achieving efficient air traffic movement in order to minimize delays. In today's airspace, human air traffic controllers control the flight trajectories of the planes. Each controller is responsible for a sector (i.e. a section of the airspace) and advises the trajectories of the planes in that sector. The main bottleneck in increasing the capacity of the airspace in the current air traffic control system is the cognitive limits of the human air traffic controllers (Erzberger, 2004). Increasing the loads of the controllers by assigning more planes to a sector increases the risk of accidents. On the other hand, reducing the sizes of the sectors in order to increase the airspace capacity is no longer feasible. Controllers need enough airspace to maneuver the aircraft; hence, sector sizes cannot be reduced beyond a certain limit. Also, decreasing the sector sizes increases the controller workload at the sector intersections.

The future air traffic control systems will need to rely on automation to increase the capacity of the airspace while preserving the high levels of dependability required in this area. Researchers at NASA proposed a vision for automating the decision-making in air traffic control system called the Automated Airspace Concept (Erzberger, 2001; Erzberger, 2004). The basic architecture of the Automated Airspace Concept is shown in Figure 1. The automated Airspace Concept gives the responsibility of determining conflict free trajectories for aircraft to a software system. Controllers are still able to interact with the system to propose changes to the trajectories. The system receives data from the radar feeds that track the aircraft. Controllers interact with the system using a graphical user interface (GUI). The system stores the four-dimensional trajectories of the aircraft in a database. The automated trajectory server accesses the trajectory database to analyze and update the trajectories.

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 contractors in making the software components highly dependable (DOT, 1998). The Advanced Automation System that was being developed during 1980s and 1990s was an ambitious effort by the Federal

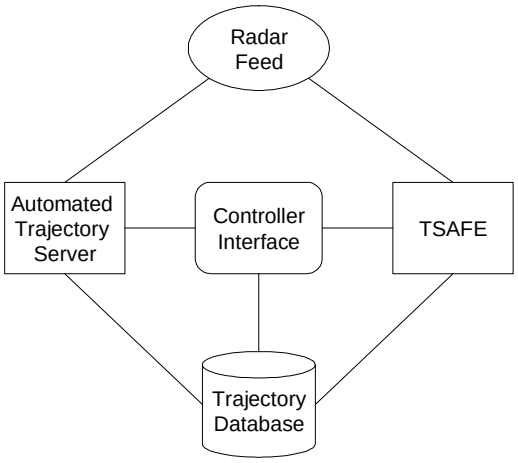

Figure 1. Automated Airspace Concept.

Aviation Administration (FAA) to replace the computer hardware and software in air traffic control facilities while increasing the level of automation. After sustaining serious cost and schedule problems, the FAA dramatically restructured the program and terminated its major segments. One of the main problems in this project was the inability of the contractors to make the Advanced Automation System software highly dependable.

To avoid a similar fate, the designers of the Automated Airspace Concept at NASA introduced a *failsafe short term conflict detection component* in their Automated Airspace Concept, which is responsible for detecting conflicts in flight paths of the aircraft within 1 minute from the current time. If a short term conflict is detected, this component takes over the trajectory synthesis function from the automated trajectory server 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.

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 (Dennis, 2003). Later on, as a part of NASA's High Dependability Computing Project, TSAFE software was integrated into an experimental environment at the Fraunhofer Center for Experimental Software Engineering, Maryland (Lindvall et al., 2005). 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.

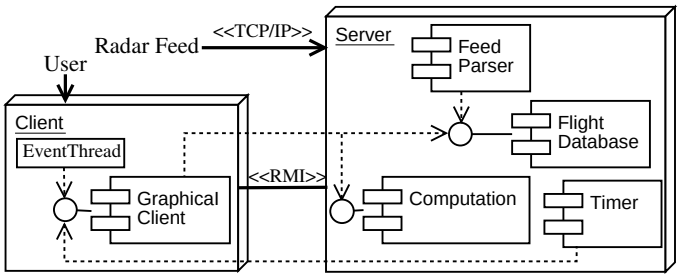

Figure 2. TSAFE architecture.

In our experimental study, we used a distributed client-server version of TSAFE, 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 conforming to the planned route or blundering).

The TSAFE III implementation consists of 21,057 lines of Java source code in 87 classes. Figure 2 shows the architecture of TSAFE III. The server component 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 side 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.

3. Concurrency Controllers

In this section we will discuss the concurrency controllers used in reengineering of the TSAFE. We will use these examples to explain the concurrency controller design pattern and the behavior of concurrency controller classes written using this design pattern. The formal semantics for the concurrency controllers is provided in (Betin-Can and Bultan, 2006).

The flight database in TSAFE is accessed by multiple threads. The integrity of the data stored in the flight database can be lost if the threads accessing the database are not properly synchronized. As an example consider the following scenario. 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. In such a scenario, an aircraft's location may be displayed in-

correctly on the client GUI. Since the client GUI provides the interaction between the human air traffic controllers and the TSAFE system, the consequences of displaying incorrect information on it could be disastrous. To prevent such synchronization problems in Java programs, Java programmers declare the methods of shared classes to be `synchronized`. However, this is not an efficient solution. 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 read-write lock. Using a read-write lock, multiple readers can access a shared resource at the same time, but a writer can access the shared resource only alone. In order to implement this solution in Java, a programmer 1) has to write a class implementing the read-write lock, and 2) needs to make sure that the appropriate methods of the read-write lock class are called before accessing the database. The design for verification approach we present below helps developers in eliminating faults in both of these two steps.

In this design for verification approach, programmers use the concurrency controller design pattern and write a set of guarded commands describing the synchronization policy without using any of the error-prone Java synchronization statements. The Java synchronization statements appear only in the predefined helper classes provided by the concurrency controller pattern, and they are automatically optimized to improve the performance.

Reader-Writer Controller: Using the concurrency controller pattern, the reader-writer synchronization policy can be implemented as a controller class. A typical implementation of the RW controller would have one integer variable (`nR`) denoting the number of readers in the critical section, one boolean variable (`busy`) denoting if there is a writer in the critical section, and four guarded commands defining four actions `w_enter`, `w_exit`, `r_enter`, and `r_exit` as shown in Figure 3. These four actions form the public methods of the RW controller class which will be called by the threads to synchronize their access to a shared resource. In the RW controller there is one guarded command for each action; however, the concurrency controller pattern allows declaration of multiple guarded commands for each action (which is necessary if different updates have to be executed based on different conditions). In the figure, the implementation of the last action of RW shows the mechanism to declare multiple guarded commands although the `r_exit` action has only one guarded command.

When an action is called, one of the enabled guarded commands of that action is executed. If none of the guarded commands of an action is enabled (i.e. all the guards evaluate to false), then the behavior is different for *blocking* and *nonblocking* actions. For example in the RW controller shown in Figure 3

```
class RWController implements ReaderWriter {
   int nR; boolean busy;
   Action act_w_enter, act_w_exit, act_r_enter, act_r_exit;
   public RWController() {
      act_w_enter = new Action (this, new GuardedCommand() {
         public boolean guard() {return (nR == 0 && !busy);}
         public void update() {  busy = true; } });
      act_w_exit = new Action(this, new GuardedCommand() {
         public boolean guard() { return true; }
         public void update() { busy = false; } });
      act_r_enter = new Action(this, new GuardedCommand() {
         public boolean guard() { return (!busy); }
         public void update() {nR = nR+1; } });
      Vector 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 );
      nR=0;
      busy=false;
   }
   public void w_enter(){ act_w_enter.blocking();}
   public boolean w_exit(){ return act_w_exit.nonblocking() ;}
   public void r_enter(){ act_r_enter.blocking();}
   public boolean r_exit(){ return act_r_exit.nonblocking();}
}
```

Figure 3. Reader-Writer controller implementation.

actions `w_enter` and `r_enter` are specified as blocking actions and actions `w_exit` and `r_exit` are specified as nonblocking actions. When a thread calls a blocking action, if all the guards are false, then the thread waits until it is notified by another thread. A nonblocking action does not cause the calling thread to wait. If all the guards are false, a nonblocking action just returns `false`, and returns `true` otherwise. The caller of a nonblocking action can determine if the action was executed or not by looking at this return value.

Note that, the developers are not required to implement the above semantics in Java in order to write a concurrency controller. This semantics is already implemented in the helper classes provided in the concurrency controller pattern. For example the helper class `Action` shown in Figure 4 should be used as is without any modifications (Betin-Can and Bultan, 2004; Betin-Can and Bultan, 2006). The `Action` class implements the semantics of the action execution as shown in Figure 4. In addition to `Action` class the `GuardedCommand` Java interface and the `StateMachine` class are provided with the concurrency controller pattern, and the developers should use them as is without modifying them. To implement a controller, the developer only writes the guarded commands of the actions as shown in Figure 3 and specifies if the actions are blocking or nonblocking. This information that needs to be provided by the programmer is shown as italic in the figure.

```
public class Action{
 protected final Object owner;
 private final Vector gcV;
 public Action(Object owner, Vector gcs){...}
 public Action(Object owner, GuardedCommand gc){...}
 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 4. Action class

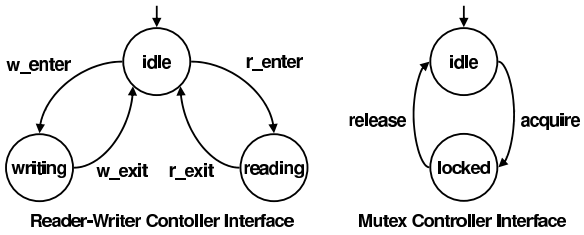

Figure 5. Controller interfaces.

One concern in using the concurrency controller pattern could be the efficiency of the synchronization based on the `Action` class shown in Figure 4. In order to address the performance concerns, we automatically optimize the concurrency controllers using a source-to-source transformation (Betin-Can and Bultan, 2004). The optimized controller class 1) uses the specific notification pattern (Cargill, 1996), 2) does not have any inner classes, and 3) minimizes the number of method invocations.

The final step in the implementation of a controller is the declaration of its interface. The interface of a concurrency controller defines the acceptable call sequences to the public methods (i.e. actions) of the controller by a thread

that uses the controller. These allowed call sequences are specified using the finite state machine implementation provided in the helper classes of the concurrency controller pattern.

A controller interface is a Java class which has the same set of methods as the controller itself. When a method of a controller interface is called, it first executes an assertion which checks that the current interface state is a state where the corresponding action can be executed, and then updates the current interface state according to the corresponding transition of the interface state machine.

The interfaces of the two concurrency controllers we used while reengineering TSAFE are shown in Figure 5. The interface of the RW controller has three states: `IDLE`, `READING`, and `WRITING` with `IDLE` being the initial state. The interface state machine shows how the interface state changes when an action is executed. The RW controller interface, for example, states that a thread using the RW controller can execute (i.e. call) the `w_exit` action only after executing the `w_enter` action.

Note that, the interface machines shown in Figure 5 are not used as synchronization mechanisms. The synchronization policy is specified in the controller classes as shown in Figure 3. The interface machines just specify how each single thread should interact with the synchronization policy. In particular, the interface machine for a controller shows the order a thread should call the controller actions.

Another crucial point is that the interface machines show the acceptable call sequence for a *single* thread. For example, the interface machine for the RW controller shown in Figure 5 states that a thread cannot execute the `r_enter` action back to back without executing a `r_exit` action in between. I.e., this synchronization policy is not reentrant. Of course other threads are allowed to execute the `r_enter` action while there is another thread in the `READING` state. This comes from the synchronization policy specified by the controller shown in Figure 3.

As we stated above, the interface machines shown in Figure 5 do not specify the synchronization policy and they are not shared among multiple threads. One can think of them as auxiliary variables that are defined to specify the acceptable call sequences for each thread. There is one separate interface machine instance for each thread. I.e., interface machine instances are not shared among multiple threads, each thread has its own local interface machine instances. During interface verification we use these local interface machine instances to check for interface violations in each thread separately as discussed in Section 5 (see Figure 13).

The controller interface is also used to specify when the methods of the shared data objects can be executed. For example, for the RW controller, a method which updates the shared data can only be executed in the `WRITING` state, a method which reads the shared data can be executed in the `READING`

```
class MutexController implements Mutex {
   boolean busy;
   Action act_acquire, act_release;
   public MutexController() {
      act_acquire = new Action (this, new GuardedCommand() {
         public boolean guard() {return (!busy);}
         public void update() { busy = true; } });
      act_release = new Action(this, new GuardedCommand() {
         public boolean guard() { return busy; }
         public void update() { busy = false; } });
      busy=false;
   }
   public void acquire(){ act_acquire.blocking();}
   public void release(){ act_release.blocking();}
}
```

Figure 6. Mutex controller implementation.

and `WRITING` states, and no method of the shared data can be executed in the `IDLE` state. In the concurrency controller pattern, these constraints are specified as assertions in a data stub class. In Section 3.1, this usage of controller interfaces is explained and exemplified with a data stub class shown as `RuntimeDatabase_Stub` in Figure 7.

Mutex Controller: The other synchronization policy used in TSAFE is the mutual exclusion (mutex) lock. One scenario where this lock is used in TSAFE is maintaining a consistent list of TSAFE clients. The clients subscribe to and unsubscribe from the TSAFE server through RMI calls. Through these calls the list of clients is accessed concurrently. This list is also accessed by another thread. The TSAFE server notifies each subscriber at certain time intervals. This mechanism is implemented in TSAFE through a `Timer` event which is dispatched by the `EventThread`. Using the concurrency controller pattern, the mutex synchronization to protect this client's list can be implemented as a controller class. Figure 6 shows a possible implementation. This implementation has one boolean variable (`busy`) denoting if there is another thread in the critical section, and two actions `acquire` and `release`.

3.1. 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 objects they are used to protect. 2) We developed the concurrency controllers implementing the synchronization policies required for accessing these shared objects. 3) We replaced all the synchronization statements in the TSAFE code with calls to the appropriate concurrency controller classes. All the synchronization state-

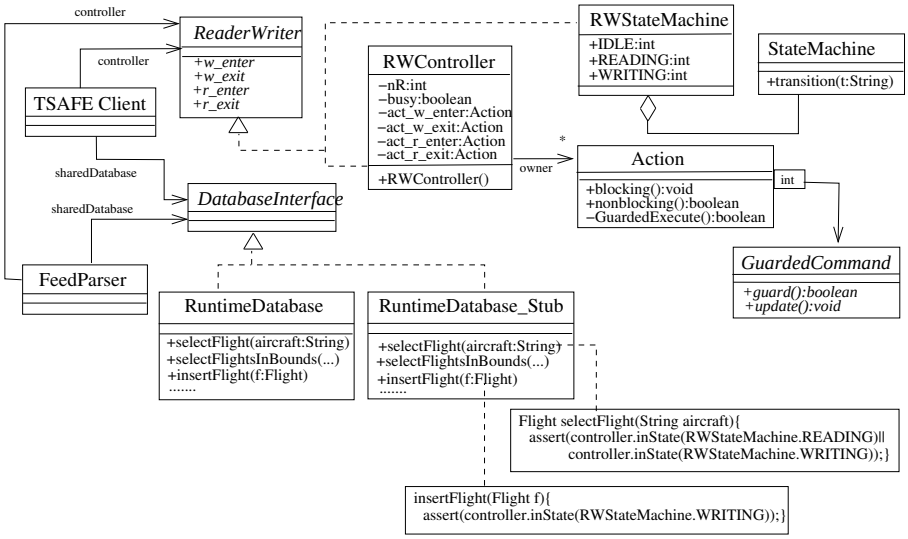

Figure 7. Synchronization of the Flight Database in TSAFE using the Reader-Writer Controller based on the Concurrency Controller Pattern.

ments in the reengineered TSAFE code are in the helper classes provided by the concurrency controller pattern.

In the reengineered TSAFE code there are two concurrency controller classes which are the RW controller and the MUTEX controller described above. There are 2 instances of the RW controller and 3 instances of the MUTEX controller protecting 6 shared objects.

Figure 7 shows a class diagram for a part of the reengineered TSAFE code where the access to flight database is protected using the RW controller based on the concurrency controller pattern. The `ReaderWriter` is a Java interface which defines the names of the controller actions. The `RWController` class contains the guarded commands specifying the controller behavior and the `RWStateMachine` class is the controller interface.

The `RuntimeDatabase` is the implementation of the flight database in TSAFE (see Figure 2). The methods of the `RuntimeDatabase` class were synchronized in the original version. Figure 7 shows two of these methods: `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. The class `RuntimeDatabase_Stub` specifies the constraints on accessing shared data based on the interface states of the RW controller. Note that, the shown assert statements imply that a thread has to call `w_enter` before calling `insertFlight` and it has to call `w_enter` or `r_enter` before calling `selectFlight`.

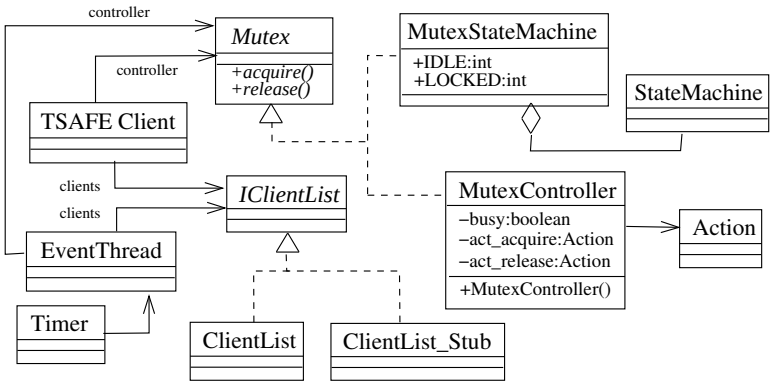

Figure 8. Use of the Mutex controller in TSAFE.

Figure 8 shows a class diagram for another part of reengineered TSAFE code. Here, the access to the client list is protected using the MUTEX controller based on the concurrency controller pattern. In this diagram, the `Mutex` is the Java interface defining the names of the controller actions, the `MutexController` is the Java class that contains the guarded commands, and the `MutexStateMachine` is the Java class realizing the controller interface. This concurrency controller protects the list of clients shown as `IClientList` in Figure 8.

4. Behavior Verification

Based on the concurrency controller pattern we divide the verification of the concurrent programs with respect to synchronization errors into two steps (Betin-Can and Bultan, 2004; Betin-Can and Bultan, 2006): 1) *Behavior verification:* Verification of the properties of the controller classes assuming that the user threads adhere to the constraints of their instances of the controller interfaces; 2) *Interface verification:* Verification of the threads which use the concurrency controllers to make sure that they access the methods of the controllers and the shared data objects in the order specified by the controller interfaces and the data stubs.

We use the Action Language Verifier (ALV) (Bultan and Yavuz-Kahveci, 2001; Yavuz-Kahveci et al., 2005) for behavior verification. We automatically translate the concurrency controllers written based on the concurrency controller pattern into the Action Language (Betin-Can and Bultan, 2004).

The Action Language is a specification language for reactive software systems (Bultan, 2000; Bultan and Yavuz-Kahveci, 2001). An Action Language specification consists of integer, boolean and enumerated variables, parameterized integer constants, and a set of modules and actions which are composed using synchronous and asynchronous composition operators. A

module specification starts with variable declarations. A state of an Action Language specification corresponds to a valuation of all the variables in the specification. The initial expression of a module defines the set of initial states of that module. Actions of a module are specified after the initial expression.

Actions of a module are written as logical expressions on primed and unprimed variables. In an action expression primed variables (called *next-state variables*) denote the next-state values and unprimed variables (called *current-state variables*) denote the current-state values. An action expression is written using current and next-state variables, arithmetic and relational expressions and logical connectives `and`, `or`, and `not`.

In the Action Language, the top level module is always called the `main` module. A module expression (which starts with the name of the module) defines the transition relation of the module. A module expression can be written as a composition of submodules and actions. Actions and modules can be composed using asynchronous and synchronous composition operators. Asynchronous composition of two actions a_1 and a_2, denoted a_1 `|` a_2, is defined as the disjunction of their transition relations. However, an action preserves the values of the variables which are not modified by itself.

ALV supports only integer, boolean and enumerated types. This means that we have to restrict the controller variables to these types in order to verify them with ALV (we use static integers as enumerated variables in the controller implementations). Since variables of the concurrency controllers 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.

The automatically generated Action Language specifications for the RW and MUTEX controllers are shown in Figures 9 and 10, respectively.

In these specifications the controller variables are defined in the `main` module. The initial values of these variables are obtained from the constructor of the controller class. Each instance of the the `RWController` module corresponds to one thread. The enumerated variable (`pc`) keeps track of the thread state which is represented by a state of the controller interface (this is the only state information we need to keep about a thread while we are verifying the controller behavior).

The controller behavior is represented with a set of actions. Each of these actions are generated from the guarded command definitions in the constructor of the controller class and from the controller interface. Both in the RW and the MUTEX controllers there is one guarded command per action. In these specifications, the nonblocking actions have been translated to two actions in Action Language. The translation of the `r_exit` action is represented with `r_exit_0` and `r_exit_1` in Figure 9. The first one represents the execution when the guard is satisfied, and the second one represents the execution when the guard is not satisfied. I.e., the guard of `r_exit_1` is the negation of the

```
module main()
 integer nR; boolean busy;
 module RWController()
  enumerated pc {IDLE, READING, WRITING};
  initial: nR = 0 and busy = false and pc=IDLE;
  r_enter_0: pc=IDLE and !busy and nR'=nR+1 and pc'=READING;
  r_exit_0: pc=READING and  true and nR'=nR-1 and pc'=IDLE;
  r_exit_1: pc=READING and !(true) and pc'= IDLE;
  w_enter_0: pc=IDLE and (nR=0 and !busy) and busy'=true
             and pc'=WRITING;
  w_exit_0: pc=WRITING and true and busy'=false and pc'=IDLE;
  w_exit_1: pc=WRITING and !(true) and pc'= IDLE;
  RWController:r_enter_0 | r_exit_0 | r_exit_1 | w_enter_0
               | w_exit_0 | w_exit_1;
 endmodule
 main: RWController() | RWController() | RWController() |
       RWController() | RWController() | RWController() |
       RWController() | RWController();
endmodule
```

Figure 9. Automatically generated Action Language specification for the RW controller.

```
module main()
 boolean busy;
 module MutexController()
  enumerated pc {IDLE, LOCKED};
  initial: busy = false and pc=IDLE;
  acquire_0: pc=IDLE and !busy and busy'=true
             and pc'=LOCKED;
  release_0: pc=LOCKED and busy and busy'=false
             and pc'=IDLE;
 MutexController:acquire_0 | release_0;
 endmodule
 main: MutexController() | MutexController() | MutexController() |
       MutexController() | MutexController() | MutexController() |
       MutexController() | MutexController();
endmodule
```

Figure 10. Automatically generated Action Language specification for the MUTEX controller.

guard of `r_exit_0`. Note that, since the guard of the non-blocking action `r_exit` is never false, the action `r_exit_1` is never executed, hence action `r_exit_1` is redundant. However, in general, for a non-blocking action we need to generate an extra action with a negated guard which corresponds to the case where the guard evaluates to false (hence the negation of the guard evaluates to true).

The Action Language translation of the RW controller shown in Figure 9 contains 8 instantiations of the `RWController` module that are composed using asynchronous composition. The 8 instantiations of the `RWController` module correspond to 8 concurrently executing threads that execute the controller actions concurrently and modify the shared variables `nR` and `busy`.

```
module main()
 integer nR; boolean busy;
 parameterized integer numInstance;
 module RWController()
  integer IDLE, READING, WRITING;
  initial: nR = 0 and busy = false;
  initial: IDLE=numInstance and READING = 0 and WRITING = 0;
  restrict: IDLE + READING + WRITING=numInstance
            and numInstance>=0;
  r_enter_0: IDLE>0 and !busy and nR'=nR+1
             and READING'=READING+1 and IDLE'= IDLE-1;
  r_exit_0: READING>0 and true and nR'=nR-1
            and IDLE'=IDLE+1 and READING'= READING-1;
  r_exit_1: READING>0 and !(true) and IDLE'= IDLE+1
            and READING'= READING-1;
  w_enter_0: IDLE>0 and (nR=0 and !busy) and busy'=true
             and WRITING'=WRITING+1 and IDLE'= IDLE-1;
  w_exit_0: WRITING>0 and true and busy'=false
            and IDLE'=IDLE+1 and WRITING'= WRITING-1;
  w_exit_1: WRITING>0 and !(true) and IDLE'= IDLE+1
            and WRITING'= WRITING-1;
  RWController:r_enter_0 | r_exit_0 | r_exit_1 | w_enter_0
               | w_exit_0 | w_exit_1;
 endmodule
 main: RWController();
endmodule
```

Figure 11. Action Language specification generated with automated counting abstraction for the RW controller.

ALV is an infinite state model checker which verifies or falsifies (by generating counter-example behaviors) CTL properties of Action Language specifications with unbounded integer variables (such as nR). For the infinite state systems that can be specified in Action Language, model checking is undecidable. Hence, ALV uses conservative approximations techniques during verification. There are three outcomes when one uses ALV to verify a system: 1) ALV verifies the property which means that the property is provably correct, 2) ALV generates a counter-example which means that the property is provably incorrect, and 3) ALV states that it is unable to verify or falsify the property. The goal of the heuristics used in ALV are to minimize the third outcome as much as possible. The undecidability of the model checking problem for ALV implies that the fixpoint computations are not guaranteed to converge. ALV uses several conservative approximation heuristics to achieve convergence. For the experiments we conducted in this study ALV was able to verify or falsify all the instances. I.e., all the fixpoint computations converged and the approximations were precise enough to verify or falsify the given properties.

Since ALV allows unbounded integer variables, it enables us to use an automated abstraction technique, called counting abstraction, to verify the concurrency controllers with respect to arbitrary number of threads (Yavuz-

```
module main()
 boolean busy;
 parameterized integer numInstance;
 module MutexController()
  integer IDLE, LOCKED;
  initial: busy = false;
  initial: IDLE=numInstance and LOCKED = 0;
  restrict: IDLE + LOCKED=numInstance and numInstance>=0;
  acquire_0: IDLE>0 and !busy and busy'=true
             and LOCKED'=LOCKED+1 and IDLE'= IDLE-1;
  release_0: LOCKED>0 and busy and busy'=false
             and IDLE'=IDLE+1 and LOCKED'= LOCKED-1;
  MutexController:acquire_0 | release_0;
 endmodule
 main: MutexController();
endmodule
```

Figure 12. Action Language specification generated with automated counting abstraction for the MUTEX controller.

Kahveci and Bultan, 2002). The counting abstraction technique (Delzanno, 2000) in ALV supports verification of parameterized systems with an arbitrary number of finite state modules. 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. The specifications generated by the counting abstraction for the RW and the MUTEX controllers are shown in Figure 11 and Figure 12, respectively.

Note that, the local variable that encodes the thread state is replaced with a set of integer variables, one for each state of the thread (i.e., one for each state of the controller interface). For example, in the parameterized specification, the integer variable `IDLE` denotes the number of threads in the interface state `IDLE`. The initial states and the transition relation of the parameterized system is defined using linear arithmetic constraints on these variables. A parameterized integer constant, `numInstance`, denotes the number of threads. This parameterized constant is restricted to be positive and 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) for ACTL properties. If an ACTL property is violated, then this means that there exists a valuation for the parameterized constant (i.e, there exists a specific number of threads) for which the property is violated and a counter-example path is generated.

Controller Properties: In order to verify the controllers with ALV we need a list of properties that specify the correct behavior of the controllers, i.e., we need the class invariants for the controller classes. We allow the CTL properties for the controllers to be either inserted directly to the generated Ac-

Table I. RW Controller Properties

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	$AG(pc = WRITING \Rightarrow AF(pc = IDLE))$
RP6	$AG(\neg(pc1 = READING \wedge pc2 = WRITING))$
RP7	$EF(pc1 = READING \wedge pc2 = READING)$
RP8	$AG(\neg(pc1 = WRITING \wedge pc2 = WRITING))$
RP9	$AG(pc1 = READING \Rightarrow nR > 0)$
RP10	$AG(pc1 = WRITING \Rightarrow busy)$
RP11	$AG(WRITING > 0 \Rightarrow AF(WRITING = 0))$
RP12	$AG(\neg(READING > 0 \wedge WRITING > 0))$
RP13	$AG(\neg(WRITING > 1))$
RP14	$AG(READING = nR)$
RP15	$AG(WRITING = 1 \Leftrightarrow busy)$
RP16	$\forall x\ AG(READING = x \wedge READING > 0 \Rightarrow AF(READING \neq x))$

tion 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 I. The properties RP1–4 only refer to the variables of the `RWController` class. For example, the global property RP1 states that whenever `busy` is true `nR` must be zero. The remaining properties refer to both the variables of the controller and also to the states of the threads. Note that the representation of the thread state is different in the concrete and the parameterized Action Language specifications. The properties RP5–10 are for concrete specifications and refer to concrete thread states. For example the property RP5 states that whenever a thread is in the `WRITING` state it will eventually reach the `IDLE` state. The properties RP11–16 are for the parameterized instances and refer to the integer variables which represent the number of threads in a particular state. For example property RP14 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 I contain universally quantified integer variables. We are able to check such properties using ALV by declaring the universally quantified variables as parameterized constants.

The properties for the MUTEX controller are shown in Table II. The properties MP1 and MP2 only refer to the variables of the `MutexController` class. The remaining properties refer to both the variables of the controller and the states of the threads. The properties MP3–6 are for concrete specifications and refer to concrete thread states. The properties MP7–10 are for the

Table II. MUTEX Controller Properties

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	$AG(LOCKED > 0 \Rightarrow busy)$
MP8	$AG(\neg busy \Rightarrow IDLE > 0)$
MP9	$AG(LOCKED > 0 \Rightarrow AF(IDLE > 0))$
MP10	$AG(!(LOCKED > 1))$

Table III. Controller Property Classification

Referring to the variables of RW controller only	RP1–RP4
Referring to concrete thread states and RW controller	RP5–RP10
Referring to parameterized thread states and RW controller	RP11–RP16
Referring to the variables of MUTEX controller only	MP1–MP2
Referring to concrete thread states and MUTEX controller	MP3–MP6
Referring to parameterized thread states and MUTEX controller	MP7–MP10

parameterized instances generated with counting abstraction and refer to the integer variables `IDLE` and `LOCKED` which represent the number of threads in the interface states `IDLE` and `LOCKED`, respectively.

5. Interface Verification and Thread Isolation

Interface verification assures that each thread conforms to the interfaces of the concurrency controllers and shared data it uses. These interfaces encode the assumptions about the thread behaviors that were used during behavior verification. The behavior verification assumes that 1) threads execute the actions of the concurrency controllers in the order defined by the controller interfaces, and 2) threads access shared data objects only at the allowed interface states of the concurrency controllers as specified in the data stubs. An example for the first assumption is that, during the behavior verification of the MUTEX controller, each thread is assumed to invoke the `release` action only after invoking the `acquire` action (see Figure 5). An example for the second assumption is that, the `EventThread` should access `IClientList` only at the `LOCKED` state of the controller interface of the MUTEX controller that is

controlling the access to `IClientList`. The goal of interface verification is to verify (guarantee) that these assumptions are satisfied by all threads.

If a thread does not violate the above two assumptions, then we say that the thread conforms to the controller interfaces. (Formal definition of interface conformance is given in (Betin-Can, 2005).) During interface verification, we check the interface conformance of each thread individually. I.e., we do not have to worry about the thread interleavings during interface verification. During behavior verification synchronization properties are verified on all possible thread interleavings that conform to the controller interfaces.

During interface verification we verify each thread in the program separately using the Java Path Finder (JPF) (Visser et al., 2003). If a thread invokes the actions of a controller in an order that is not allowed by the interface of that controller, then the thread does not conform to the controller interface. For example, in a program using the MUTEX controller with the controller interface shown in Figure 5, the following sequence will cause an interface violation: Thread T_1 invokes `acquire`, thread T_1 invokes `release`, thread T_2 invokes `release`, ... The violation here is in thread T_2 since there is no transition originating from the initial interface state with the `release` action. On the other hand, the thread T_1 conforms to the controller interface of MUTEX on part of the program trace shown above. If a thread invokes a method of a shared data item at a controller interface state that is different than the ones specified in its data stub, then the thread does not conform to the controller interface. As explained in Section 3, the methods of the controller interfaces and data stubs have assertions to check these criteria. If JPF reports a violation of an assertion in a controller interface or a data stub, then we know that the thread in question caused an interface violation. JPF outputs a counter-example execution trace that leads to the violation of the assertion.

During interface verification we have to keep track of the interfaces of all the controllers used by a thread. In fact, we check whether each thread obeys an interface state machine that is the product of all the interface state machines of all the controllers used by that thread. By default, this product machine allows a thread to execute an action of a controller only when the thread is in the initial states of the interface state machines of all the other controllers it is using (Betin-Can and Bultan, 2006). For example, if a thread is using both an RW and a MUTEX controller, then in order to execute an action of the RW controller, it has to be in the `IDLE` state of the MUTEX controller. Although this default policy was sufficient for TSAFE, it may not be suitable for some other applications. For example, it may be necessary to modify two shared data items protected by different controllers at the same time. There are two ways of handling such situations in our framework. The first option is to merge the behaviors of the two concurrency controllers and use one concurrency controller to protect access to both shared data items. The second option is to define a new product machine (that is different than the default

one) by composing the interfaces of the two controllers as discussed in (Betin-Can and Bultan, 2003). The interface verification technique we discuss below works for both of these options and also for the default product machine we used during the interface verification of TSAFE.

Our interface verification approach is thread modular, i.e., we check each thread separately for interface violations. For this purpose, we isolate each thread by a conservative approximation of the behavior of other threads in the distributed program without modifying the thread code. To explain our modular interface verification and thread isolation techniques, we first introduce a simple model for distributed programs. Then, we present our thread isolation techniques for conservatively approximating the environment of a thread in Sections 5.1 and 5.2.

A distributed program $DP = \{P_1, P_2, ..., P_k\}$ is a set of local programs running on different machines (in Java, different JVMs), where k is the number of machines. We assume that the local programs communicate with remote procedure calls. In TSAFE, these remote procedure calls are performed with Java RMI.

Each program P_i consists of a set of threads, i.e., $P_i = \{T_1, T_2, ..., T_{n_i}\}$ where n_i is the number of threads in P_i. There are three types of threads in a program: 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`. In TSAFE, there are two types of implicitly created threads: the event thread (T_{Ev}) that dispatches the GUI events and the threads created to serve RMI calls (T_{RMI}).

The threads communicate with each other through a shared store. Based on the concurrency controller pattern, the shared store contains shared data objects and controller objects. In addition to the shared store, each thread has a local store which is accessed only by that thread. Each thread also has a control state which represents the value of its program counter. The state of a thread T is represented with the shared store, the local store, and the control state of T, i.e., $State_T \in Shared \times Local(T) \times Control(T)$.

A thread execution $e = op_0, op_1, \ldots$ is a sequence of operations. A thread can perform two kinds of operations: local operations which only change the thread's local store and control state, and interaction operations through which the thread interacts with its environment. The interaction operation types are 1) a read operation from the shared store, 2) a write operation to the shared store, 3) RMI operation, 4) GUI operation, 5) file read and write operations, 6) socket operation, and 7) thread creation operation. Another form of environment interaction is through input events. The input events are GUI events (e.g. button click event) and the incoming RMI events from other remote programs. (We name the outgoing RMI call an RMI operation and the

incoming RMI call an RMI event.) We isolate a thread from its environment by modeling these interaction operations and input events.

5.1. Modeling Interaction Operations with Stubs

In modeling interaction operations, we use the transformations shown in Table IV. We discuss these transformations below. The transformations in the first two rows transform operations on shared store to assertions on local controller interface state for detecting interface violations. We are able to apply these two transformations since the elements of shared store are known at verification time. These elements, which are controllers and shared data protected by these controllers, are implemented explicitly in a program implemented based on the concurrency controller pattern.

The first transformation on Table IV abstracts the interaction through a concurrency controller. Let T be a thread that accesses controller C. We apply this transformation for each controller action act of C. The transformation abstracts the internals of the controller action act (conditional waits, guarded commands, etc.) with the action sequencing rules T has to obey. This transformation enables us to detect interface violations as follows. Let $CI = (Q, q_0, A, R)$ be the controller interface of C. (Here Q is the set of interface states, $q_0 \in Q$ is the initial state, A is the set of controller actions, and $R \subseteq Q \times A \times Q$ is the transition relation. Although we can handle nondeterministic interfaces, here, to simplify the discussion, we assume that there is at most one next state for each state and action pair.) Based on concurrency controller semantics (Betin-Can, 2005), whenever the control state of T is at a controller action act of C, T updates its current interface state ($\texttt{cur} \in Q$) according to the transition relation R. (Note that, `cur` is a local variable of T, i.e., as we mentioned above, we are checking interface conformance separately for each thread.) The assertion in the transformed operation checks whether such transitions are defined. If the assertion fails, then T does not conform to the controller interface CI and an interface violation has occurred. To perform this transformation, we replace the controller C in the program with CI and create one CI instance per thread. Note that, although the instances of C are shared, the instances of CI are not shared. Controller interfaces are in the local store of the thread and any interface operation only influences the $Local$ of the thread (i.e., `cur` is a local variable of T as mentioned above). Therefore, the concurrency controller is removed from the $Shared$ (see Figure 13).

The second transformation on Table IV abstracts the interaction through shared data. Let $sho \in Shared$ be a shared data. The transformation abstracts the internals of the read or write operation on sho with rules of accessing sho and a conservative approximation of the influence of other threads through sho. In this transformation, generated assert statements check the access rules

Table IV. Transformations to model interaction operations

Interaction type	Interaction [a] operation	Substitution
Controller action [b]	$C.act()\{body\} : r$	$CI.act()\{$ `assert` $(\exists q' \in Q, (\mathtt{cur}, act, q') \in R)$; `cur` $:= q'$; `return choose`(r); $\} : r$
read/write on $sho \in Shared$ [c]	$m_s(a_0, a_1, \ldots, a_l)$ $\{body\} : r, E$	$m'_s(a_0, a_1, \ldots, a_l)\{$ `assert` $(\mathtt{cur} \in Q_m)$; `if` (`choose(bool)`) `then return choose`(r); `else throw choose`(E); $\} : r, E$
RMI operations	$remote.m_{RMI}(a_0, \ldots, a_l)$ $\{body\} : r, E$	$remote'.m_{RMI}(a_0, ..., a_l)\{$ `if` (`choose(bool)`) `then return choose`(r); `else throw choose`(E); $\} : r, E$
operations 4–7 [d]	$m(a_0, \ldots, a_l)\{body\} : r, E$	$m'(a_0, ..., a_l)\{$ `if` (`choose(bool)`) `then return choose`(r); `else throw choose`(E); $\} : r, E$

[a] $body$ denotes the method body, $a_0, a_1, .., a_l$ is the argument sequence of the method, r is the return type of the method, and E is the set of exception types that can be thrown by the method.

[b] $CI = (Q, q_0, A, R)$ is the interface of the concurrency controller C, where Q is the set of interface states, $q_0 \in Q$ is the initial state, A is the set of controller actions, $R \subseteq Q \times A \times Q$ is the transition relation, and $act \in A$.

[c] m_s is a read or write operation on sho, and Q_m is the set of allowed interface states to invoke method m_s.

[d] m is an operation of type 4–7.

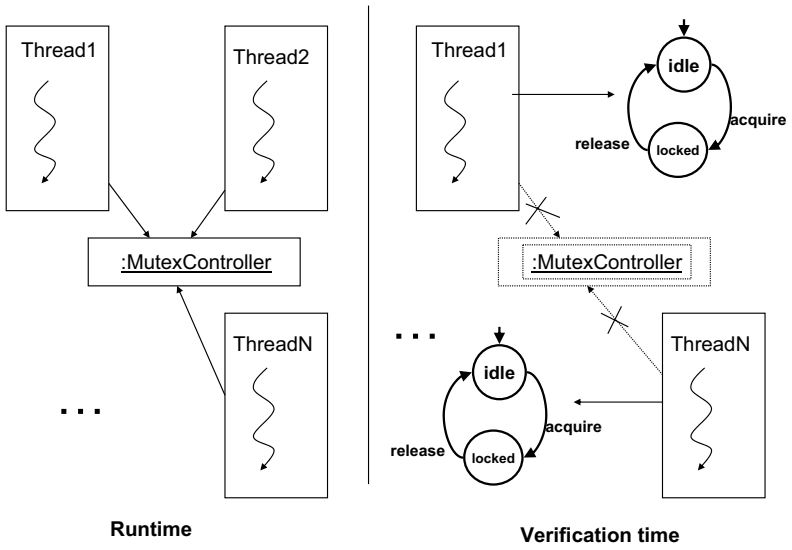

Figure 13. Local instances of interface machines are used as stubs for concurrency controllers during interface verification.

of sho. These assertions check whether the thread in question (T) performs a read or write operation on sho at an allowed interface state of CI, where CI is the interface of the concurrency controller protecting sho. To reflect the effect of the other threads on T through sho, we use the nondeterminism function `choose`. The effects occur either through the return value of the method invoked to perform a read or write operation on sho, or through the exceptions thrown by that method. In the Table IV, this method is denoted as m_s. By using the nondeterminism function, we approximate the return value of m_s and the exceptions thrown by m_s. The function `choose` returns a nondeterministically chosen value of its argument type and if its argument is a set of types, a type is chosen nondeterministically before returning a nondeterministic value of that type. (The values to be chosen nondeterministically are decided based on the data dependency analysis which is explained in Section 5.3.) During the interface verification, each of these nondeterministic choices are explored; hence, the stub methods conservatively approximate the influence of other threads on T. The realization of this transformation is as follows. Based on the concurrency controller pattern, the assertion and the allowed interface states are specified by the programmer in the data stub for

sho. Therefore, the abstraction of the shared data is achieved by replacing the shared data classes with their corresponding shared data stub classes.

These substitutions, however, are not sufficient to isolate a thread for some distributed programs such as TSAFE. The steps above only abstract the interaction operations 1 and 2. Below we explain how we model the rest of the interaction operations.

An RMI operation is a method call on a remote object. In Table IV, this operation is shown as $remote.m_{RMI}(a_1, a_2, .., a_l)\{body\} : r, E$ where $remote$ denotes the remote object and m_{RMI} denotes the remote method with an argument sequence $a_1...a_l$, return type r, and exception type set E. Therefore, we identify the remote objects and their remote methods before applying the transformation. When a thread T performs an RMI operation, it is only affected by the return value and the exceptions thrown during the remote call. The result of the transformation for RMI operations in Table IV conservatively approximates these effects in the stub method with the nondeterminism function. We substitute each remote object $remote$ with a local stub object $remote'$. This stub object includes a stub method for each remote method of $remote$. In addition, since the stub $remote'$ is a local object that resides in the local store, we also abstract the remote procedure call details.

Transformations for the rest of the interaction operations follow the same principles. The interaction operation performed by the thread T with a method call is replaced with a stub method call that overapproximates the return value and exceptions thrown. This transformation is shown in the last row of Table IV. Unlike the remote methods, the interaction methods for 4–7 are predetermined. For example, in Java, the graphical methods are in libraries such as the `java.awt` or `javax.swing` library. Therefore, we use one-time-implemented stub methods while transforming such operations. I.e., for these operations we do not have to create new stubs for each program, we can use the same generic stubs for every program. (We assume that any graphical method outside the graphical library eventually reaches a method within the library. If this assumption does not hold, e.g. the thread code implements the actual bit placement on the screen, stubs for such methods should be generated.)

For Java programs, we achieve the above abstractions for interaction operations 3–7 by stub class substitutions. A stub for a class contains every accessible method declaration of that class. The methods of a stub class are the results of the transformations discussed above. We implemented `choose` with the JPF's nondeterminism utilities `Verify.random(int)` and `Verify.randomBool()`. These utilities force JPF to exhaustively search for every possible choice. Therefore, at verification time, the nondeterminism in the code results in an exhaustive search, *not* in random testing. We developed generic stubs for the file, socket, and GUI operations; and we automatically

generate stubs for RMI operations. Since JPF is only able to handle pure Java, we also replace all native calls with stubs.

5.2. Modeling Thread Initialization and Input Events with Drivers

Drivers are necessary to transform each thread execution into a standalone program execution. Recall that, there are three types of threads in a Java program: the main thread, explicit threads, and implicit threads. We create different types of drivers for each thread type. The driver of the main thread nondeterministically assigns values to the command line arguments using the nondeterminism utilities of JPF. The drivers for the explicitly created threads simulate the thread creation by assigning nondeterministic values during the initialization of the thread. In other words, these drivers set the initial configuration for $Shared$ and $Local$.

Here we discuss the drivers for implicit threads. Such drivers model the execution of an implicit thread by producing all possible sequences of input events related to that kind of thread. In TSAFE, the implicitly created threads are the event thread that dispatches the GUI events and the threads created to serve RMI calls. When a Java program has GUI objects, a thread (called `EventDispatchThread`) is instantiated implicitly. The event thread, denoted by T_{Ev}, captures the user interactions which are the GUI events and invokes the corresponding event listeners. An RMI thread, denoted as T_{RMI}, is responsible for serving the incoming RMI calls. We generate a driver for these implicit threads that overapproximates their executions. The driver of T_{RMI} creates all possible event sequences, $e = ev_1, ev_2, \ldots, ev_j$ for all $j \geq 0$ where ev_i is an element in the set of remote events created by other programs. Similarly, the driver of the event thread T_{Ev} overapproximates the execution of T_{Ev} by creating all possible GUI event sequences.

We generate the Java code for the drivers of implicit threads automatically. A generated driver code consist of an initialization block and an event loop which simulates the input event sequences. At each iteration of this loop one input event is chosen nondeterministically by using `Verify.random(int)` and JPF will search all possible event sequences if this loop is infinite. However, most of the time JPF runs out of memory if we leave this loop as an infinite loop. Hence, the user has to limit the number of iterations. Note that, sometimes JPF is able to search the whole state space even for an infinite event sequence since the state space may be finite.

Driver for Event Thread: The initialization block of the driver for an event thread T_{Ev} has two responsibilities. Firstly, this block launches the GUI components. Secondly, it finds all the visible and enabled GUI objects that have registered event listeners. The second part of the driver, which is the event loop, simulates the event thread. This loop is the essential part that simulates

the behavior of the event thread. At each iteration, the driver first chooses one of these GUI objects, chooses an event, and calls the listeners for that event in this order.

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 in the initialization block is created by copying the relevant parts from the application code. The generator identifies all possible user event types, i.e. the GUI events, by finding all different event listener types in the code.

Driver for RMI Threads: In Java, the RMI calls that a program receives are the invocations of the methods whose signatures are defined in the remote Java interface that extends `java.rmi.Remote`. Whenever an RMI call is received, it is directed to an instance of the concrete class that implements this remote Java interface. The driver we generate simulates this behavior.

The driver of an RMI thread T_{RMI}, in the initialization block, instantiates the concrete class implementing the remote interface. If the concrete class looks up another RMI component (i.e. if there is a call to `Naming.lookup(String)` method), the initialization block of the driver should register the RMI stub of that component to the `Naming` class. In the event loop, the driver produces all possible remote event sequences and directs these RMI input events to the concrete class instance.

Our automated RMI thread driver generator inspects the source code of the remote interfaces to collect the RMI events. During this inspection, the generator finds the remote method signatures in the source code and uses them to create RMI events. The generator also inspects the source code of the concrete class to synthesize a piece of code for the aforementioned initializations.

If we try to verify a thread with respect to all possible inputs from its environment provided by generic or automatically generated stubs and drivers, typically, JPF runs out of memory. We provide a data dependency analysis to identify the input parameters that may influence the thread behavior with respect to the interface correctness conditions. Using the results of this data dependency analysis, the user is expected to restrict the domains of the input parameters.

5.3. Data Dependency Analysis

It is possible that some of the input parameters (or the return values) that are passed to a thread via drivers or stubs may not influence the synchronization behavior of a thread. We are only interested in the influences that might lead to an interface violation. Hence, we implemented a data dependency analysis to identify the input parameters affecting the synchronization behavior.

The analysis consists of multiple backward traversals on the program dependence graphs (Ottenstein and Ottenstein, 1984). The starting point of each traversal is determined as follows. For each method in the program, if there are branching statements that determine whether a concurrency controller or a shared data method is called or not, 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 backwards and the visited definition sites are collected. The visited statements are marked to avoid entering infinite loops. The result of this procedure is a backward dependence tree per starting point. The vertices are the definition sites collected during the traversal. The leaves are the influencing input parameters. A path in the tree shows how these input parameters control the execution of the shared operations.

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 the same field. For example, there is an implicit dependency between `get` methods and `set` methods of the same class. A get method that returns the value of a field is implicitly dependent on the set method that assigns a value to the same field.

We implemented this analysis using the Soot Java optimization framework (SOOT, 2005), which uses a 3-address representation for Java. The analysis determines which statements, directly or indirectly, affect the reachability of invocation of a method to perform a shared operation. 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 the set of methods updating each class field (and updating the elements of that class field). We represent this set as DEF(*field*) where *field* denotes a class field. Then we examine each method in detail. First, by using the control flow graph of the method, we find all the branch conditions that determine whether the controller or shared data stub method is called. If such a branch condition exists, we find a backward dependency tree by using the findDependents function shown in Figure 14. We send the branching statement and the method we are analyzing as the actual parameters for this function. This function is a recursive backward dependence tree construction.

In the algorithm given in Figure 14 the function def$(m)(var, stmt)$ denotes the definition analysis utility of Soot. This function returns the definition sites for the variable var at statement $stmt$ in method m. The function use$(stmt)$ returns the variables that are used on the right hand side of the statement

```
findDependents(Method m,Statement stmt)
returns t:Tree
 t.root:=stmt
 foreach var ∈ use(stmt)
   t.addChild(findDependentsH(m,var,stmt))
 end for
 foreach branching statement b effecting stmt
   t.addChild(findDependents(m,b))
 end for
 return t


findDependentsH(Method m,Variable var,Statement stmt)
returns t:Tree
 t.root:=var
 foreach stmt' ∈ def(m)(stmt, var)
   foreach u ∈ use(stmt')
     if u is a local variable then
       t.addChild(findDependentsH(m,u,stmt'))
     end if
     if u is a class field then
       foreach m' ∈ DEF(u)
         if m' has not been visited for u then
           foreach exit statement stmt'' of m'
             t.addChild(findDependentsH(m',u,stmt''))
           end for
       end for
       t.addChild(findDependentsH(m,u,stmt'))
     end if
     if u is a parameter of m then
       foreach (m', m) ∈ CallGraph
         if m' has not been visited as a callee then
           foreach statement stmt'' of m' that is a call to m
             t.addChild(findDependentsH(m',u,stmt''))
           end for
         end if
       end for
     end if
   end for
   if stmt' is a call site of some method m' then
     t.addChild(findDependentsH(m',var',r))
       where r is the return statement of m' and var' is the local used in r
   end if
 return t
```

Figure 14. findDependents Algorithm

$stmt$. The CallGraph denotes the call graph of the program where an edge (m', m) denotes a call from m' to m.

The analysis results are used in the construction of drivers and stubs. For the input parameters that do not influence the synchronization behavior, a constant value with the correct type is given in the driver or the stub implementations. For the ones that might influence the synchronization behavior

there are two possibilities. If the domain of the input parameter is finite (e.g. boolean) we enumerate all possible values and choose one value using JPF's nondeterminism utilities during verification. Otherwise, the analysis results are inspected and an influencing value set is provided by the user. During the execution of drivers or stubs, the value of the input parameter is chosen from this predetermined value set with the `Verify.random(int)` utility of JPF. It is possible to automate this process with a theorem prover such as (Paulson, 1994), or using the techniques described in (Visser et al., 2004; Xie et al., 2005) for the input parameters of heap type.

We have tried to use the slicers available in the Bandera (Dwyer et al., 2001) and Indus (Indus, 2005) toolsets in our dependency analysis. In our experiments, however, we found that both of these tools, at the time this paper was written, failed to capture the *implicit* interprocedural dependencies described above, e.g., of some `getField` from its corresponding `setField`.

5.4. Isolating Threads of TSAFE

TSAFE is composed of a client component and a server component. The client component interacts with a user through GUI objects. This component also interacts with the server component through RMI calls. The server component interacts with many client components through RMI. This component also gets radar input feeds through a TCP/IP connection. The server also interacts with its environment through timer events. In this section we discuss the isolation of the threads in client and server components.

5.4.1. *Client Component*

The TSAFE's client component 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 server component. The implicitly created threads are the event thread and the RMI thread that serves the remote calls initiated by the server component.

The environment of the main thread contains only GUI component stubs and a stub for the `java.rmi.Naming` class. We provide these generic stubs as a part of our framework, i.e., they are used as is without any modification by the user. Using these stubs is the application of the transformation shown in the last row of Table IV. However, there is some user intervention necessary for the environment modeling of the event thread and the RMI thread as explained below.

The Event Thread: We have isolated the event thread with a driver and using the transformations for the GUI operations shown in Table IV. The transformations for GUI operations are applied through the GUI stubs provided by our framework. The driver for the event thread is generated automatically as

discussed in Section 5.2 and data value assignments are performed using the results of the data dependency analysis.

In the end, the generated event thread driver simulates 4 different event types to be directed to the instances of 9 different GUI classes. The event types are ListSelectionEvent, ActionEvent, ItemEvent, and WindowEvent. Some of the GUI classes to respond to the events are FlightList, GraphicalWindow and TsafeMenu.

The RMI Thread: The TSAFE's client component has 2 RMI operations and 4 RMI events. We have generated one stub to model the RMI operations to be used by the server component and one driver to model the RMI thread in the client component automatically. The generated RMI stub implements the stub methods for the 2 RMI operations. This RMI stub is the stub class shown as $remote'$ in Table IV. The generated RMI driver for the TSAFE client registers an RMI stub of the TSAFE server component and instantiates the client component in the initialization block. In the event loop, the driver produces all possible event sequences with the 4 input events.

5.4.2. *Server Component*

The server component has two implicitly created threads, a main thread, and an explicitly created thread. The implicitly created threads are the RMI thread 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.

The event thread in the server component has two responsibilities. The first one is to prepare and launch TSAFE. Since this task does not involve concurrency, we have omitted these operations while creating the environment of the event thread. The second responsibility of this thread is to handle the events created by a timer. Therefore, the initialization block of the event thread driver finds the `Timer` object, and the event loop calls its registered listener.

To isolate the RMI thread at the server component we have applied the techniques discussed in Section 5.2. However, due to the launch mechanism in the server component and our objective of not modifying the application code during interface verification, we have inserted a piece of code that finds the launch button and sends a click event into the RMI driver.

The feed parser thread is created at launch by the event thread. We have separated this thread creation operation interaction with the stub substitu-

tion discussed Section 5.1. In the following discussion, we explain how we isolated the feed parser thread.

The Feed Parser Thread: The feed parser thread is isolated from its environment by 1) a driver that initializes its local and shared store and 2) interaction operation models. In this section, we explain the socket operation model tailored for Java programs. The principle in this model is the same as the general stub model.

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`.) We model this behavior for TCP clients as follows. First, we replace the `Socket` with an empty stub. Then, we model reading streams from a socket through a `BufferedReader` (or `Reader`) stub. This stub returns one of the possible string values whenever the program requests data. Currently, these values are inserted in the stub code based on the results of the data dependence analysis. For UDP communications, programs read packets from a `DatagramSocket` via a `DatagramPacket`. We model this behavior by using an empty stub for `DatagramSocket` and a `DatagramPacket` stub which returns one possible byte array value. Currently, the set of possible byte array values are added in the stub code based on the dependence analysis. Finally, sending data for both communication types is modeled via the empty stubs of `OutputStream` for `Socket` and `DatagramSocket`, respectively.

In TSAFE, the feed parser thread 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 above. In this model, the contents of the messages are determined by the data dependency analysis. The analysis results have showed that among 5 attributes of a message received from the feed source only the characters denoting the message type attribute (the first attribute) and the exceptions thrown during the socket operations affect the synchronization behavior.

6. Experiments

In this study, our goal was to experimentally evaluate the effectiveness of the design for verification with concurrency controllers technology in finding synchronization errors in safety critical air traffic control software. For this purpose, we followed an approach similar to mutation testing which is used for measuring the effectiveness of a test set (Kim et al., 2000; Budd, 1981; Ammann et al., 1998; DeMillo et al., 1978). In mutation testing, first,

a number of slight variations of a program are generated. Then, the effectiveness of a test set is measured by examining if the test set can distinguish these variations, called mutants, from the original program. In our experiments, we have created slight variations of TSAFE by fault seeding and examined whether our technique can capture these faults.

We performed two sets of experiments based on fault seeding. During the first study, 40 different versions of TSAFE were created by manual fault seeding. The first set explored the types of faults that can be verified using the presented design for verification approach.

We performed a new set of experiments as an extension to our conference paper (Betin-Can et al., 2005b) to evaluate the presented verification techniques in a larger scale. During this experimental study, we implemented a random fault injection program based on the fault categorization of the first study. We created 100 faulty versions of TSAFE automatically and verified each version using the presented modular verification technique. Automated, random fault seeding enabled us to generate a larger number of faulty versions of TSAFE compared to manual fault seeding. Moreover, automated fault seeding eliminated the possible human bias from the fault seeding process and resulted in a repeatable experimental setup that can be used by other researchers in the future.

In this section, we first explain the setup for both experiments and the types of faults. Next we present the results of the manual fault injection experiment and provide a more detailed analysis compared to our conference paper (Betin-Can et al., 2005b). Then we proceed with the results of the new study and present a discussion about the results of our experiments.

6.1. Experimental Setup and Fault Seeding

During the experimental study, the authors were divided into 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.

Before the experiments, the UCSB team reengineered the TSAFE software as described in Section 3 and generated the drivers and the stubs for thread isolation as explained in Section 5. During this reengineering process, we found a synchronization error in TSAFE where a shared object used for RMI connection was not protected by any synchronization statements. The reengineering of the TSAFE software using the concurrency controllers was done in 8 hours by one team member (5.5 hours for the server component and 2.5 hours for the client component). During the thread isolation process, the inspection of the results of the data dependency analyses for the refactored version of TSAFE took 13 minutes. Based on these inspection, restricting

the input domains used in the drivers and stubs took 18 minutes. Originally, TSAFE consisted of 87 classes with 21,057 lines of Java code.

Manual fault seeding set-up: In the manual fault injection study, 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. Recall that we are focusing on errors that might occur while using the concurrency controller pattern. We are not trying to categorize errors in an arbitrary concurrent program.

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, and 4) *blocking/nonblocking faults* (CB) which were created by making a nonblocking action blocking or visa versa.

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, and 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, and 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 V 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.

Automated random fault seeding set-up: In the study with automated random fault injection, we implemented a random fault seeding program to generate modified versions of TSAFE based on the fault types discussed above. In the reengineered TSAFE, there were 21 places to insert controller faults. To insert the interface faults, we used the classes that had references to concurrency controller or shared data instances. There were 7 classes (1373 SLOC) of this kind. Each modified version generated by this program contained at most four faults. The number of faults and the type of faults were selected randomly for each version. The mutant generator did not check whether there was an equivalent mutant. All of the mutants generated compiled success-

Table V. Faulty versions

Type	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

fully. We created 100 modified versions of TSAFE and applied our modular verification technique. There were 30 CI faults, 38 CG faults, 41 CU faults, 33 IM faults, 22 ICV faults, and 24 ICN faults.

The automated fault injection program works as follows. The IM faults are injected by adding, removing or swapping controller methods before or after a shared data access by the program. To introduce an ICV fault, the program chooses an integer or boolean class field and inserts a conditional using this field before a controller method call statement. The ICN faults are injected by creating a new unique integer variable, and adding a conditional statement using this new variable before a method call to a controller. This new variable is initialized to zero and is incremented every time the control reaches the inserted conditional statement. The conditional is of the form `if (_new_var00 < C)` where `C` is a constant integer value that can be 50, 60, 70, 80, 90, 100, 200, 300, 400, or 500. The purpose of using different constant values is to evaluate whether there is a limit to the depth of faults we can discover and whether we can generalize our results.

The controller faults are inserted as follows. To inject a CI fault, the program chooses a controller variable and inserts an assignment statement at the end of the controller's constructor method. The program initializes boolean variables to a randomly chosen boolean constant and the integer variables to a randomly chosen integer value between -5 and +5. The fault injection for CG is performed by first selecting a guard expression, which is the statement inside one of the `public void guard()` methods in the constructor of the controller class. We change either a variable or an operator that occurs in the guard expression with one of its possible alternatives randomly. The possible alternative of an integer variable is another integer variable or an arithmetic expression. The possible alternative of an operator is another operator of the same type. For example, the relational operator `>=` can be replaced with another relational operator such as `==`. If the guard expression is a constant, such as the guard expression of the `r_exit` action, then a boolean expression

is generated using the controller variables (e.g. `nR==5 && busy`). The fault injection for CU is performed in a similar manner where either an operator or a variable is replaced with a possible alternative.

In terms of mutation testing, our random fault seeding program uses the following mutation operators to generate mutants: 1) add a controller method call, 2) remove a controller method call, 3) swap controller methods before or after a shared data access 4)insert a conditional on an existing integer field before a controller method call, 5) insert a conditional on an existing boolean field before a controller method call, 6) insert a conditional using a new integer variable before a controller method call, 7) insert assignments to controller fields in the controller's constructor, 8) replace an integer variable in a guard or update clause with another integer field, 9) replace an integer variable in a guard clause with an arithmetic expression, 10) change an arithmetic operator in a controller class, 11) change a relational operator in a controller class, 12) change a logical operator in a controller class, 13) replace the constant guard expression with a boolean expression.

6.2. RESULTS

In this section, we first present the results of the manual fault seeding study where the authors have worked as two teams. Then we present the results of automated random fault seeding experiments.

6.2.1. *Study1: Manual Fault Seeding*

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 V, 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. The modification in v5 (see Figure 15) changed the `release` action in the MUTEX controller from blocking to nonblocking. This modification does not change the behavior of the controller since the guard of the `release` action is `true`, i.e., it never blocks. The modification in v13 (see Figure 16) changed an assignment in the `acquire` action of the MUTEX controller from `busy=true` to `busy=!busy`. However, this modification does not cause any failures since `busy` is always `false` when the `update()` of `acquire` is called.

Table VI. Controller property violations

Type	Version	Violated Property
CI	v2	RP1, RP3, RP6, RP9, RP11, RP12, RP14, RP16
CI	v4	MP1
CG	v3	MP2, MP8
CG	v6	RP1, RP4, RP5, RP6, RP8, RP10, RP11, RP12, RP16
CU	v7	RP6, RP9, RP11, RP12, RP16
CU	v13	no error (spurious)
CU	v14	type error at MutexController, caught by translator
CU	v16	RP2
CU	v24	RP1, RP3, RP6, RP9, RP11, RP12, RP16
CU	v25	RP6, RP9, RP11, RP12, RP16
CB	v5	no error (spurious)
CB	v21	MP2
CB	v28	MP3, MP4, MP5, MP6, MP7, MP8, MP10
CB	v34	MP1, MP3, MP9

```
(1) public void release() {
(2)   act_release.blocking();}
```
(a) Original code

```
(1) public void release() {
(2)   act_release.nonblocking();}
```
(b) Code with seeded fault

Figure 15. Original and fault seeded code segments for the spurious controller fault in version v5.

```
(1) gcs.add(new GuardedCommand() {
(2)    public boolean guard() {
(3)      return (!busy);}
(4)    public void updates() {
(5)      busy = true;}
(6)    });
(7) act_acquire = new Action(this, gcs);
```
(a) Original code

```
(1) gcs.add(new GuardedCommand() {
(2)    public boolean guard() {
(3)      return (!busy);}
(4)    public void updates() {
(5)      busy = !busy;}
(6)    });
(7) act_acquire = new Action(this, gcs);
```
(b) Code with seeded fault

Figure 16. Original and fault seeded code segments for the spurious controller fault in version v13.

To show which properties were more effective in discovering faults, Table VI gives the controller faults and the properties from Table I and Table II that were violated by these faults. The controller properties MP1 and MP2 were crucial for finding the seeded faults in the MUTEX controller. However, these two controller properties were not sufficient for finding all the seeded faults in the MUTEX controller. For example, the seeded fault in the version v28 does not cause violation of the properties MP1 and MP2. Note that, all the properties of the MUTEX controller that were violated by the seeded fault in version v28 refer to the thread states. The properties MP1 and MP2, on the other hand, do not refer to the thread states; they only refer to the variables of the controller. In order to find all the seeded faults it was necessary to have *at least* one property that refers to the thread states.

The properties RP6, RP9, RP11, RP12, and RP16 were the most effective group of properties for finding the seeded faults in the RW controller. Apart from the seeded fault in version v16, all the seeded faults in the RW controller result in violation of at least one of these properties. The only property that is violated by the seeded fault in version v16 is RP2. Interestingly, none of the other seeded faults lead to violation of the property RP2.

Interface verification using JPF caught 22 of the 26 seeded interface faults. Table VII lists the 26 versions, the type of the seeded interface fault for each version, and the violation caused by the seeded fault that was found during the interface verification. For example, the violation in v1 is an access to the `RuntimeDatabase` object at an illegal `RWController` state. To be specific, in this version, the Server-RMI thread invokes the `selectFlightsInBounds` method of the `RuntimeDatabase` while the thread is in the `IDLE` state of the RW controller. This interface violation occurred because the seeded fault was the removal of the `r_enter` method of the RW controller.

Among the seeded interface faults, one of them (v33) that was not caught by JPF was a spurious fault. However, the faults in versions v18, v19, and v20 were real faults which can cause interface violation but were not found by JPF. We will discuss these faults in detail in Section 6.3.

As aforementioned, the seeded interface fault in v33 was a spurious fault. Figure 17(a) shows the original code fragment and Figure 17(b) shows the seeded interface fault in v33. The seeded fault in v33 is the addition of the if-else-statement at lines (4.1) and (4.2). Originally, there was only the `mutex_stopped.release();` statement instead of these lines. This modification does not cause an error since the MUTEX lock is always released after accessing the shared data as it was originally implemented.

Table VIII shows the performances of ALV and JPF. The first part of the table shows the performance of ALV during behavior verification for different controller instances and the second part shows the performance of JPF during interface verification for different threads. The first four columns show the memory and time consumption (mean and standard deviation) for the verifi-

Table VII. Interface violations

Version	Violated interface	Type
v1	accessing RuntimeDatabase at wrong RW state	ICV
v7	wrong action sequence at RWController	IM
v8	wrong action sequence at RWController	IM
v10	wrong action sequence at RWController	IM
v11	wrong action sequence at RWController	IM
v12	wrong action sequence at RWController	ICN
v15	wrong action sequence at RWController	IM
v17	wrong action sequence at RWController	ICN
v18	not detected	ICN
v19	not detected	ICN
v20	not detected	ICN
v22	wrong action sequence at MutexController	IM
v23	accessing Bool (stop flag) at wrong Mutex state	IM
v26	accessing RuntimeDatabase at wrong RW state	ICV
v27	wrong action sequence at MutexController	ICV
v29	accessing RuntimeDatabase at wrong RW state	IM
v30	accessing RuntimeDatabase at wrong RW state	ICV
v31	wrong action sequence at MutexController	ICV
v32	accessing ClientList/Vector at wrong Mutex state	ICV
v33	no interface violation (spurious fault)	ICV
v35	wrong action sequence at RWController	ICV
v36	accessing RuntimeDatabase at wrong RW state	ICV
v37	accessing RuntimeDatabase at wrong RW state	ICV
v38	wrong action sequence at MutexController	ICV
v39	accessing RuntimeDatabase at wrong RW state	ICV
v40	accessing RuntimeDatabase at wrong RW state	ICV

```
(1) private final boolean isStopped() {
(2)   mutex_stopped.acquire();
(3)   boolean result=stopped.get();
(4)   mutex_stopped.release();
(5)   return result; }
```

(a) Original code

```
(1) private final boolean isStopped() {
(2)   mutex_stopped.acquire();
(3)   boolean result=stopped.get();
(4.1) if (false) mutex_stopped.release();
(4.2) else mutex_stopped.release();
(5)   return result; }
```

(b) Code with seeded fault

Figure 17. Original and fault seeded code segments for the spurious controller fault in version v33.

Table VIII. Verification and falsification performance for modified versions with manual fault seeding

Controller Instance	Verify (using ALV)				Falsify (using ALV)			
	Memory(MB)		Time(sec)		Memory(MB)		Time(sec)	
	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev
RW-8	3.70	0.10	2.26	0.17	3.67	0.70	1.65	1.86
RW-16	9.36	0.04	3.42	0.03	13.06	0.56	9.94	1.26
RW-P	12.05	1.01	0.21	0.02	5.48	0.28	5.21	0.05
MUTEX-8	0.22	0.00	0.02	0.00	0.20	0.02	0.02	0.00
MUTEX-16	0.62	0.01	0.02	0.00	0.60	0.01	0.03	0.00
MUTEX-P	0.24	0.01	0.03	0.00	0.28	0.00	0.03	0.00

Component-Thread	Verify (using JPF)				Falsify (using JPF)			
	Memory(MB)		Time(sec)		Memory(MB)		Time(sec)	
	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev
Client-Main	2.32	0.01	2.00	0.00	–		–	–
Client-Event	33.09	5.13	663.21	10.27	12.2	0.00	15.63	0.00
Client-RMI	40.96	3.70	25.39	4.71	42.64	0.00	10.12	0.00
Server-Main	67.72	0.01	17.08	0.00	–		–	–
Server-Event	10.95	1.84	7.57	0.98	9.56	0.00	6.88	0.00
Server-RMI	26.80	9.97	136.90	38.82	24.92	9.51	29.74	29.12
Server-Feed	83.49	30.32	123.12	49.57	94.72	93.99	18.51	10.11

cation of an instance without any property or interface violations and the last four columns show the memory and time consumption (mean and standard deviation) for counter-example generation for the instances with property or interface violations.

Table IX. Average falsification performance with respect to the fault categories in manual fault seeding

Model Checker	Type	Memory(MB)		Time(sec)	
		Mean	StDev	Mean	StDev
ALV	CI	2.33	2.60	2.80	1.60
ALV	CG	2.33	2.07	3.33	1.86
ALV	CU	0.22	0.06	2.96	0.15
ALV	CB	0.22	0.06	0.02	0.01
JPF	IM	44.45	59.16	14.43	6.91
JPF	ICV	23.50	9.84	31.85	32.25
JPF	ICN	27.21	4.51	56.80	53.54

Falsification performance with respect to the fault categories are shown in Table IX. The first part gives the time and memory consumption to falsify the TSAFE versions during behavior verification. The values are averaged over the versions that have the same type of controller fault. For example, the average time spent to falsify each of the 6 versions defected with a CU fault is 2.96 seconds. Similarly, the second part shows the average resource consumption during interface verification to falsify the instances of each of the interface fault category. For ICN faults, the memory consumption is higher than the other fault categories. The performance for the fault in version v17 appears in this group. This fault is a deep fault and the state space JPF has to search until finding the error is huge. In Section 6.3 we discuss this type of seeded faults in these experiments.

6.2.2. *Study2: Automated Random Fault Seeding*

We created 100 modified versions of TSAFE for the experimental study with automated random fault seeding. The number of faults and the type of faults were selected randomly for each version. Each modified version contained at most four faults. The random fault generator implemented for this study used the fault types determined in the manual fault seeding study. Three versions had no faults. Among the rest of the modified TSAFE versions, 75 of them had controller faults and 61 of them had interface faults. There were a total of 109 controller faults and 79 interface faults in versions vr1–vr100.

When we verified the controllers in versions vr1–100 with ALV we found 76 faults in the controllers. The 33 faults that were not found by ALV were spurious faults. Among the 33 spurious faults, 12 of them were in the RW controller. The fault seeding program changed the update statement of `w_exit` action from `busy = false` into `busy=!busy` in 4 versions. This modification did not cause any faults since `busy` was always `true` when the update of this action was invoked. A similar spurious seeded fault occurred in the `w_enter` action where `busy=true` was changed to `busy=!busy` in 3 versions. This modification did not cause any faults since `busy` was always `false` when the update of the `w_enter` action was invoked. In 2 versions, the initialization statement for the variable `busy` was changed to an expression what was equivalent to the original one. Another fault was changing the guard expression of `w_exit` action from `true` into `busy` in 3 versions. The assumption of the RW controller on the user threads prevented this change to be an error since according to the interface specification this action could be executed only if there is a preceding `w_enter` action. In other words, `busy` was always `true` whenever the guard of the `w_exit` action was invoked.

There were 21 spurious faults in the MUTEX controller versions. In one version, the guard of the the `release` action was changed from `busy` to `!!busy`. This was the result of applying the same CG fault that negates a boolean expression twice on the same guard expression. In 8 versions the up-

Table X. Controller verification and falsification performances for the modified versions with random fault seeding

Controller Instance	Verify (using ALV)				Falsify (using ALV)			
	Memory(MB)		Time(sec)		Memory(MB)		Time(sec)	
	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev
RW-8	3.01	0.80	0.16	0.09	3.05	0.95	0.18	0.17
RW-16	9.08	1.64	1.24	0.65	4.70	1.14	0.21	0.21
RW-P	3.76	0.97	0.21	0.01	17.52	8.59	0.05	0.02
MUTEX-8	0.22	0.06	0.01	0.00	0.19	0.06	0.01	0.01
MUTEX-16	0.62	0.16	0.02	0.01	0.68	0.59	0.03	0.03
MUTEX-P	0.37	0.08	0.02	0.00	0.47	0.37	0.03	0.04

date of the `release` action was changed from `busy=false` to `busy=!busy` and in 7 versions the update of `acquire` action was changed from `busy=true` to `busy=!busy`. In 5 versions, the automated fault seeding changed the initialization `busy=false` to `busy=false`.

Table X shows the ALV performance during behavior verification of the concurrency controllers in the modified versions with random fault seeding. The second column shows average memory consumption during the behavior verification of the instances with no property violations in these experiments. The sixth column shows the average memory consumption for counter-example generation for the instances with property violations. The table also shows average time spent both during the verification of the instances with no property violations and the counter-example generation for the randomly modified concurrency controller versions. The standard deviations for the time spent and the memory used both in verification and falsification are shown in columns marked with StDev.

Interface verification using JPF recognized 55 of the 61 versions with interface faults. Among the modified versions that were not recognized, 5 versions had spurious faults. In 3 versions, the errors introduced were ICV faults where an integer field is used in the branch condition. This field was a constant field and the branch condition always evaluated to true, which made the if-statement redundant. In the other 2 versions, the fault inserted were ICV faults using the boolean fields of the classes. Within the methods where the ICV faults were seeded, these variables always had the boolean value that satisfied the inserted branch condition. However, the faults in one version (vr86) were real faults that can cause interface violations. These faults were the same type of faults that were not caught in the first experiment as well (the faults v18–v20). These faults were deep faults and the state space JPF had to search to find an interface violation was huge. In Section 6.3 we discuss this type of seeded faults in both of our experiments. There were other deep faults

Table XI. Interface verification and falsification performances for the modified versions with random fault seeding

Component-Thread	Verify (using JPF)				Falsify (using JPF)			
	Memory(MB)		Time(sec)		Memory(MB)		Time(sec)	
	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev
Client-Main	3.46	0.12	1.92	0.05	-	-	-	-
Client-Event	26.39	4.68	489.62	48.47	16.74	0.00	241.35	0.00
Client-RMI	52.04	3.10	14.19	2.26	84.73	78.84	10.30	6.71
Server-Main	35.26	2.01	12.19	0.04	-	-	-	-
Server-Event	49.91	5.22	12.93	0.55	89.89	73.42	11.29	6.24
Server-RMI	25.94	10.91	78.30	5.04	29.27	8.13	12.62	13.94
Server-Feed	61.79	6.30	201.88	0.578	20.16	40.39	4.37	4.00

in this experiment. However, 7 of these deep faults occurred on the same path with a shallow fault. Since JPF stopped the execution when it caught the first interface violation, these deep faults were masked by other faults. JPF recognized the faulty version because the shallow fault caused an interface violation earlier than the deep fault. The deep faults were introduced with ICN faults. The rest of the deep faults are caught since JPF could handle the increase in the state space. In these faults, the constant used in the conditional statement inserted was less than 100, except from two ICN faults. In the first of these two, the constant used was 200, while in the second one the constant was 400 and the fault occurred within the scope of event thread of the server component. In this study, the deep faults appeared on the execution path of the RMI thread of the client component, and of the event and feed parser threads of the server component.

Table XI shows the performances of JPF during interface verification for the threads of TSAFE. The first four columns show the mean and standard deviations for the memory and time consumptions for the verification of the instances with no interface violations. The last four columns show the mean and standard deviations for the memory and time consumptions for counter-example generation for the instances with interface violations. During the falsification of the RMI thread of the client component and the event thread of the server component, the memory consumption increased since these threads had deep ICN faults. A more detailed information about the JPF performance during these experiments is shown in Figures 18 and 19. Figure 18 shows the time and memory usage during the interface verification of the randomly modified TSAFE versions that do not have an interface violation. Figure 19 shows the time and memory usage during counter-example generation for the randomly modified TSAFE versions that do have interface violations. In this figure, the outliers in three graphs are due to the ICN faults, which increase

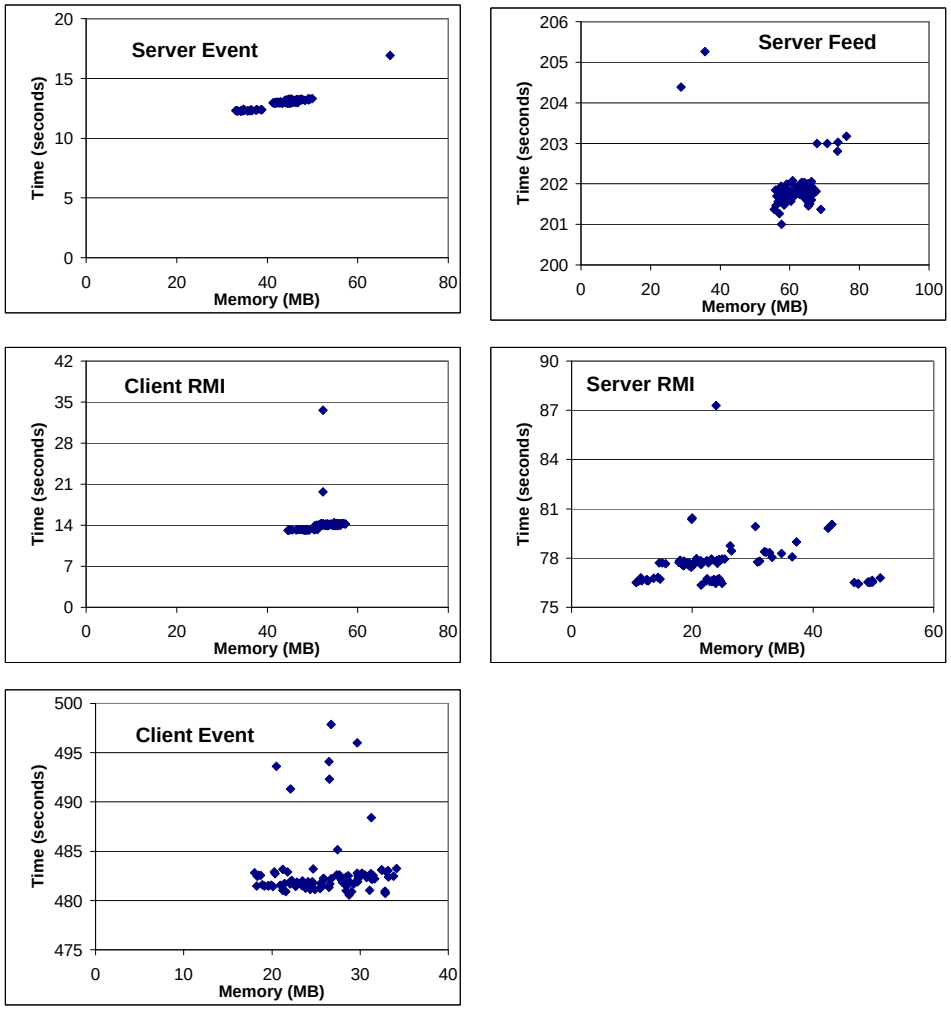

Figure 18. Interface verification performance for vr1–vr100 with JPF

the state space. However, in the graph for RMI threads of the server component, the outlier is due to an ICV fault. Although this fault is quickly caught, the number of states visited by JPF was close to the number of states visited while checking an RMI thread of the server component with no interface violations. In the falsification performance graphs, falsification of the event thread in the client component is not shown since there is only one version with an error in this thread.

Table XII shows time and memory consumption required to find an error in the faulty versions with respect to the fault categories. The values are averaged over the versions that have the same type of controller or interface fault.

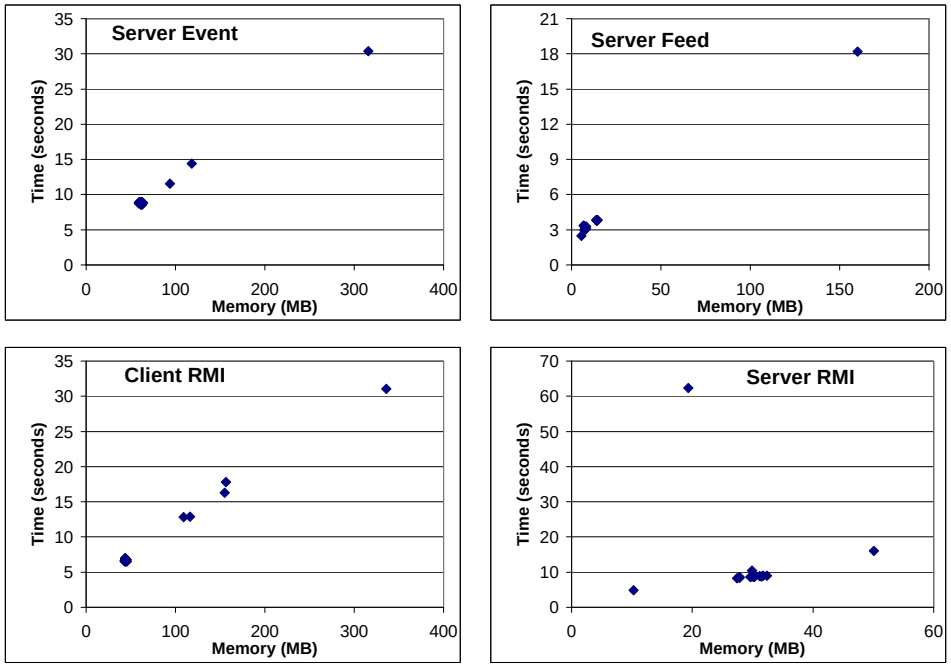

Figure 19. Interface falsification performance for vr1–vr100 with JPF

Table XII. Average falsification performance with respect to the fault categories in random fault seeding

Model Checker	Type	Memory(MB)		Time(sec)	
		Mean	StDev	Mean	StDev
ALV	CI	1.55	0.42	0.09	0.06
ALV	CG	1.60	0.51	0.10	0.10
ALV	CU	1.40	0.48	0.06	0.08
JPF	IM	30.29	20.72	13.56	40.36
JPF	ICV	46.26	19.86	7.93	1.97
JPF	ICN	135.81	100.75	15.25	8.33

These numbers show that the falsification performance during the random fault seeding was similar to the falsification performance during the manual fault seeding experiment, see Table IX.

6.3. DISCUSSION

In this section we discuss the results of our two experimental studies. Through these studies, we investigated the effectiveness of the presented modular ver-

ification technique using a similar approach to mutation testing. In our experiments, we used our fault categorization to create faulty versions. Using the terminology of the mutation testing, the experiments showed that our verification technique killed almost all the mutants (except for some of the ICN faults, which were missed due to the resource constraints of the JPF program checker). These experiments also enabled us to analyze verification performances on a real software and demonstrate the possible difficulties.

ALV Performance: For behavior verification we generated three instances of each controller: two concrete instances with 8 and 16 threads and a parameterized instance using counting abstraction (denoted with suffixes 8, 16, and P in Tables VIII and X). We checked 6 properties on both the concrete and parameterized instances of the MUTEX controller. For the RW controller we checked 10 properties on the concrete instances (RP1–10 in Table I) and 11 properties on the parameterized instance (RP1–4 and RP11–16 in Table I). Both verification and falsification of the MUTEX controller was more efficient compared to RW controller since it was a smaller specification with fewer variables.

Concrete vs. Parameterized Instances: Both verification and falsification performance for the parameterized instances are typically between the concrete cases with 8 and 16 threads. Note that, the verification results for the parameterized instances are stronger compared to the concrete cases since they indicate that the verified properties hold for an arbitrary number of threads. However, for falsification the results of the concrete and parameterized instances are equivalent, both of them generating a counter-example behavior demonstrating the fault. Note that 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 an arbitrary number of threads.

JPF Performance: Table XI and the second part of Table VIII show the performance of JPF during the interface verification of each isolated thread (see Section 5). 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

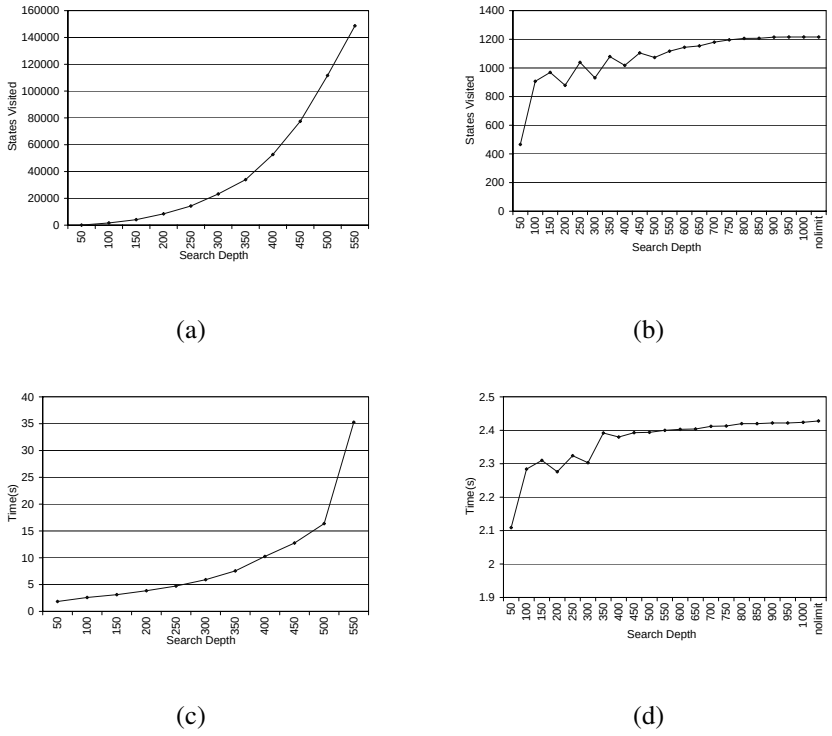

Figure 20. JPF performance using actual controller instance ((a),(c)) and instances of the controller interface as stubs ((b),(d))

used in the environment. Still, overall, falsification performance is better than the verification performance.

Our experiments on TSAFE show that our verification approach is able to handle large programs. Note that, JPF is not able to analyze TSAFE without our modular verification approach and thread isolation, and ALV is not directly applicable to verification of Java synchronization operations. Separating controllers and threads using controller interfaces improves the scalability and applicability of the model checking techniques we have used. To further demonstrate how our technique improves the scalability of verification, we performed another experiment on a small sized concurrent program. This program consisted of two threads, one shared data buffer instance, and one concurrency controller instance. The threads just access the controller and the data instance (put an item into or take an item from the buffer) in an infinite loop. We tried to verify this program 1) directly with JPF where the controller instance is not replaced with a stub, and 2) using our modular verification approach where instances of the controller interface are used as stubs for the

controller. JPF ran out of memory when we did not use our modular verification approach. Therefore, we experimented with limited search depths. While using the controller instance directly, JPF ran out of memory when the search depth is 600 or over. This problem never occurred when using stubs, since stubs have finite reachable state spaces. JPF successfully verified the problem instance with stubs without a limit on the search depth. The results are given in Figure 20. The graphs on the left are using the actual controller instances for different search depths. The graphs on the right are using interfaces as stubs for different search depths. Figure 20(a) and Figure 20(b) show the growth of unique states visited. Number of visited states grows exponentially in (a), whereas in (b) it stabilizes. Figure 20(c) and Figure 20(d) show the execution time of JPF with respect to increasing search depth.

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 developer 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 (see Section 3.1), which generates optimized controller classes using a source-to-source transformation (Betin-Can and Bultan, 2004), since this step only uses the parts verified during behavior verification.

Unknown Shared Objects: The developers may not realize that some objects are shared and therefore not use concurrency controllers to protect them. In that case, the presented verification approach will not be helpful since it only checks access to shared objects identified by the developers using the data stubs. Similar errors happen in standard Java programming when programmers do not use the Java synchronization primitives to protect access to shared objects. In fact, during the refactoring process, we found such an error in TSAFE where a shared object used for RMI connection was not synchronized. We fixed this error by introducing a mutex controller. We are working on extending 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 thread's scope and become accessible by another thread. Such analysis can be used to identify the objects which need to be synchronized. The analysis techniques we investigated so far (Bogda, 2001; Indus, 2005) either do not scale to programs as big as TSAFE or identify too many objects as shared. We think that this is a promising direction for future research.

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 controllers was all about the variables of the controllers and 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 real faults that were missed by the presented verification approach: the interface faults in versions v18, v19, and v20 in the manual fault seeding study and in version vr86 in the random fault seeding study. These faults were all ICN faults with branch conditions in front of a method call to a controller. The only difference between the faults in the versions v17, v18, v19, and v20 was the constant value in the branch conditions which was 100, 1000, 10000, and 100000, respectively. In the random fault seeding study, the constant used was 400 and the fault occurred on the execution path of RMI thread of the server component.

Interface verification with JPF identifies the fault in v17 however misses the faults in v18, v19, v20 in the first experiment. In the second experiment, JPF misses the faults in version vr86. 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 the 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 a 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 during the design phase. We use the controller interfaces to model the environments of the concurrency controllers and shared objects. Similarly, interfaces can be used for modeling the environments of threads.

These experimental observations help us to identify the strengths and weaknesses of our verification technique. One of the weakness of our technique is that it will not be helpful when not all of the shared data objects are not identified and, therefore, not protected by a concurrency controller. Another weakness is the difficulty in finding deep faults due to the program checker

used in interface verification that employs explicit state verification techniques. One positive observation about the presented technique is its ability to distinguish the spurious faults. In addition, the modularity in our technique enables us to discover the faults in different categories in a reasonable amount of time.

7. Related Work

This paper is an extended version of our conference paper (Betin-Can et al., 2005b). The concurrency controller pattern was proposed originally in (Betin-Can and Bultan, 2004). In (Betin-Can and Bultan, 2006) a formal model for the concurrency controller pattern is presented and the correctness criteria for the behavior and interface verification steps are formally defined. In (Betin-Can et al., 2005a), a related design pattern, called the peer controller pattern, is proposed for design and verification of asynchronously communicating web services. The work in (Betin-Can et al., 2005a) demonstrates that the basic principles used in the design for verification approach discussed in this paper can be extended to other domains. Our main contributions in this study are: 1) Two experimental studies demonstrating the applicability of the presented design for verification approach to safety critical air traffic control software and empirical results demonstrating the effectiveness of the modular verification strategy based on the concurrency controller pattern. One set of experiments was done using manual fault seeding and another set of experiments was done using random fault seeding. 2) Techniques for thread isolation, including a data dependency analysis and generic drivers, and stubs for modeling the environments of threads for GUI components, RMI connections and network communication. 3) A fault classification for identifying the types of faults that can be discovered by our approach. 4) Identification of the strengths and weaknesses of the verification techniques used in the presented design for verification approach, based on the observations made during the experiments. Preliminary results from this study were published in (Betin-Can et al., 2005b). This extended version contains a new set of experiments with random fault seeding, a more thorough analysis of the experimental results, the specification and discussion of an additional concurrency controller, a more detailed discussion on the TSAFE software component, and more detailed explanations of the concurrency controllers, the verification techniques, the thread isolation techniques and the dependency analysis used during thread isolation.

There have been other studies on design for verification. The approach in (Sharygina et al., 2001) focuses on verification of UML models whereas we focus on verification of programs. Use of design patterns to improve the efficiency of automated verification was also proposed in (Mehlitz and Penix,

2003). However, our interface-based modular verification technique is different than the approach presented in (Mehlitz and Penix, 2003). They suggest using design patterns for code separation, to partition a large program into independently verifiable components. On the other hand, through the concurrency controller pattern, our modular verification strategy not only separates concurrent threads but also checks the properties of their collaboration, i.e. the synchronization policy.

In (Chakrabarti et al., 2002) interfaces of software modules are specified as a set of constraints, and algorithms for interface compatibility checking are presented. In (DeLine and Fahndrich, 2004) type systems are extended with stateful interfaces and interface checking is made part of type checking. We use interfaces as part of a design pattern for concurrency controllers and use finite and infinite state model checking techniques together to verify both controller behaviors and interfaces.

Model checking finite state abstractions of programs has been studied in (Ball and Rajamani, 2001; Chaki et al., 2003; Dwyer et al., 2001). We present a modular verification approach where behavior and interface checking are separated based on the interface specification provided by the programmer. Also, using infinite state verification techniques, we are able to verify concurrency controller classes with respect to arbitrary number of threads.

In (Godefroid et al., 1998) an open reactive program is converted 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. The techniques presented in (Tkachuk and Dwyer, 2003; Tkachuk et al., 2003) generate environments for 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.

Stoller (Stoller and Liu, 2001) 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 program dependence-based abstraction selection methodology discussed in (Dwyer et al., 2001) guides the user to choose abstractions to the variables affecting the property and the control flow. This is similar to our approach in which the user inspects the analysis results and chooses appropriate valuations.

The graphical user model in (Dwyer et al., 2004) is similar to our generic GUI driver. That model, however, creates all types of user events after choosing a GUI object. The actual event thread, 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 interaction orderings.

The thread-modular reasoning discussed in (Flanagan and Qadeer, 2003) 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.

To avoid the error-prone usage of low-level synchronization primitives, the recently released J2SE 5.0 includes a concurrency utilities package. 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. Our approach to behavior verification 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.

Mutation testing and manual fault seeding are the frequently used techniques to measure the effectiveness of a test set (Kim et al., 2000; Budd, 1981; Ammann et al., 1998; DeMillo et al., 1978). For example, in (Do and Rothermel, 2006), both manual fault seeding and automated mutant generation techniques are used to compare test case prioritization techniques empirically. Memon et al. used fault seeding to evaluate an automated regression tester for GUI applications (Memon and Xie, 2005). In our experiments, we have created slight variations of TSAFE by manual fault seeding and by random mutant generation. We examined whether our technique increase the scalability of model checking and able to capture these variations.

Our fault categorization, as opposed to general fault classifications such as (IEEE, 1993), is focused on the fault types that are possible to occur when using the concurrency controller pattern. We do not consider the faults that do not affect the synchronization aspect of a concurrent program. Our fault categorization is orthogonal to other concurrency faults identified in the literature, such as the ones in (Long and Strooper, 2003). Many of those faults are due to erroneous use of the synchronization primitives and are eliminated by our pattern (e.g., premature release of locks).

8. Conclusions

In this paper we experimentally evaluated the effectiveness of the design for verification with concurrency controllers in finding synchronization errors in safety critical software. The concurrency controller pattern supports a modular verification strategy by identifying the stateful interfaces of concur-

rency controllers. Based on these interfaces, verification of the synchronization policies (implemented as concurrency controllers) are separated from the verification of their correct usage by different threads. We presented techniques for thread isolation which enables verification of each thread separately.

We reengineered an automated air traffic control software component called TSAFE using the concurrency controller design pattern. We conducted two sets of experiments based on fault seeding. First, we created 40 faulty versions of TSAFE using manual fault seeding. During this exercise we also developed a classification of faults that can be found using the presented design for verification approach. Next, we generated another 100 faulty versions of TSAFE using randomly seeded faults that were created based on this fault classification. The presented verification techniques were able to find almost all of the seeded faults. In addition to demonstrating the effectiveness of the presented design for verification approach in eliminating synchronization faults, the results of our experiments helped us identify new directions for improvement.

References

Ammann, P. E., P. E. Black, and W. Majurski: 1998, 'Using Model Checking to Generate Tests from Specifications'. In: *ICFEM '98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*.

Ball, T. and S. K. Rajamani: 2001, 'Automatically Validating Temporal Safety Properties of Interfaces'. In: *Proceedings of the SPIN Workshop*. pp. 103–122.

Betin-Can, A.: 2005, 'Design for Verification for Concurrent and distributed programs'. Ph.D. thesis, University of California Santa Barbara.

Betin-Can, A. and T. Bultan: 2003, '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)*, Vol. 89.

Betin-Can, A. and T. Bultan: 2004, 'Verifiable Concurrent Programming Using Concurrency Controllers.'. In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*. pp. 248–257.

Betin-Can, A. and T. Bultan: 2006, 'Highly Dependable Concurrent Programming Using Design for Verification'. Technical Report 2006-05, Computer Science Department, University of California, Santa Barbara, (Submitted for publication).

Betin-Can, A., T. Bultan, and X. Fu: 2005a, 'Design for Verification for Asynchronously Communicating Web Services'. In: *Proceedings of the 14th International World Wide Web Conference (WWW)*. pp. 750–759.

Betin-Can, A., T. Bultan, M. Lindvall, S. Topp, and B. Lux: 2005b, '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)*.

Bogda, J. G.: 2001, 'Program Analysis Alleviates Java Synchronization'. Ph.D. thesis, University of California, Santa Barbara.

Budd, A. T.: 1981, 'Mutation Analysis: Ideas, Examples, Problems and Prospects'. In: *Computer Program Testing*. pp. 129–148.

Bultan, T.: 2000, 'Action Language: A Specification Language for Model Checking Reactive Systems'. In: *Proceedings 22nd International Conference on Software Engineering*.

Bultan, T. and A. Betin-Can: 2005, 'Scalable Software Model Checking Using Design for Verification'. In: *Proceedings of the IFIP Working Conference on Verified Software: Theories, Tools, Experiments*.

Bultan, T. and T. Yavuz-Kahveci: 2001, 'Action Language Verifier'. In: *Proceedings 16th IEEE International Conference on Automated Software Engineering (ASE)*. pp. 382–386.

Cargill, T.: 1996, 'Specific notification for Java thread synchronization'. In: *Proceedings 3rd Conference on Pattern Languages of Programs (PLoP)*.

Chaki, S., E. Clarke, A. Groce, S. Jha, and H. Veith: 2003, 'Modular Verification of Software Components in C'. In: *Proceedings of International Conference on Software Engineering (ICSE)*. pp. 385–395.

Chakrabarti, A., L. de Alfaro, T. Henzinger, M. Jurdziński, and F. Mang: 2002, 'Interface compatibility checking for software modules'. In: *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*. pp. 428–441.

DeLine, R. and M. Fahndrich: 2004, 'Typestates for Objects'. In: *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*. pp. 465–490.

Delzanno, G.: 2000, 'Automatic Verification of Parameterized Cache Coherence Protocols'. In: *Proceedings 12th International Conference on Computer Aided Verification*, Vol. 1855 of *LNCS*. pp. 53–68.

DeMillo, R. A., R. J. Lipton, and F. G. Sayward: 1978, 'Hints on Test Data Selection: Help for the Practicing Programmer'. *IEEE Computer* pp. 34–41.

Dennis, G.: 2003, 'TSAFE: Building a Trusted Computing Base for Air Traffic Control Software, Master's Thesis, Massachusetts Institute of Technology'.

Do, H. and G. Rothermel: 2006, 'On the use of mutation faults in empirical assessments of test case prioritization techniques'. *IEEE Transactions on Software Engineering* pp. 733–752.

DOT: 1998, 'Advance Automation System'. Dep. of Transportation, Office of Inspector General, Audit Report, AV-1998-113.

Dwyer, M. B., J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, W. Visser, and H. Zheng: 2001, 'Tool-supported Program Abstraction for Finite-state Verification'. In: *Proceedings of International Conference on Software Engineering (ICSE)*. pp. 177–187.

Dwyer, M. B., Robby, O. Tkachuk, and W. Visser: 2004, 'Analyzing Interaction Orderings with Model Checking'. In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*. pp. 154–163.

Erzberger, H.: 2001, 'The Automated Airspace Concept'. In: *Proceedings of USA/Europe Air Traffic Management R&D Seminar*.

Erzberger, H.: 2004, 'Transforming the NAS: The Next Generation Air Traffic Control System'. In: *Proceedings of the 24th International Congress of the Aeronautical Sciences*.

Flanagan, C. and S. Qadeer: 2003, 'Thread-modular Model Checking'. In: *Proceedings of the SPIN Workshop*. pp. 213–224.

Godefroid, P., C. Colby, and L. Jagadeesan: 1998, 'Automatically Closing Open Reactive Programs'. In: *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. pp. 345–357.

IEEE: 1993, 'IEEE Standard Classification for Software Anomalies'. IEEE Standard 1044-1993.

Indus: 2005, 'Indus'. http://indus.projects.cis.ksu.edu.

Kim, S., J. Clark, and J. McDermid: 2000, 'Class mutation: Mutation testing for object-oriented programs'. In: *Proceedings of the FMES*.

Lea, D.: 1999, *Concurrent Programming in Java*. Reading, Massachusetts: Addison-Wesley.

Lindvall, M., 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: 2005, 'An Evolutionary

Testbed for Software Technology Evaluation'. *NASA Journal of Innovations in Systems and Software Engineering* **1**(1), 3–11.

Long, B. and P. Strooper: 2003, 'A Classification of Concurrency Failures in Java Components'. In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS03)*.

Mehlitz, P. C. and J. Penix: 2003, 'Design for Verification Using Design Patterns to Build Reliable Systems'. In: *Workshop on Component-Based Software Engineering*.

Memon, A. M. and Q. Xie: 2005, 'Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software'. *IEEE Transactions on Software Engineering* pp. 884–896.

Neumann, P. G.: 2004, 'Risks to the Public in Computers and Related Systems'. *ACM Software Engineering Notes* **29**(4), 7–14.

Ottenstein, J. and L. M. Ottenstein: 1984, 'The Program Dependence Graph in a software Development Environment'. *ACM Software Engineering Notes* pp. 177–184.

Paulson, L. C.: 1994, *Isabelle: A Generic Theorem Prover*, Vol. 828 of *Lecture Notes in Computer Science*.

Sharygina, N., J. C. Browne, and R. P. Kurshan: 2001, 'A Formal Object-Oriented Analysis for Software Reliability: Design for Verification'. In: *Proceedings of Fundamental Approaches to Software Engineering (FASE)*. pp. 318–332.

SOOT: 2005, 'Soot: a Java Optimization Framework'. http://www.sable.mcgill.ca/soot/.

Stoller, S. D. and Y. A. Liu: 2001, 'Transformations for Model Checking Distributed Java Programs.'. In: *Proceedings of the SPIN Workshop*. pp. 192–199.

Tkachuk, O. and M. B. Dwyer: 2003, 'Adapting Side-Effects Analysis for Modular Program Model Checking'. In: *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*. pp. 188–197.

Tkachuk, O., M. B. Dwyer, and C. Pasareanu: 2003, 'Automated Environment Generation for Software Model Checking'. In: *Proceedings of the 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*. pp. 116–129.

Visser, W., K. Havelund, G. Brat, and S. Park: 2003, 'Model Checking Programs'. *Automated Software Engineering Journal* **10**(2), 203–232.

Visser, W., C. S. Pasareanu, and S. Khurshid: 2004, 'Test input generation with Java PathFinder'. In: *Proceedings of International Symp. on Software Testing*.

Xie, T., D. Marinov, W. Schulte, and D. Notkin: 2005, 'Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution'. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

Yavuz-Kahveci, T., C. Bartzis, and T. Bultan: 2005, 'Action Language Verifier, Extended'. In: *Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005)*. pp. 413–417.

Yavuz-Kahveci, T. and T. Bultan: 2002, 'Specification, Verification, and Synthesis of Concurrency Control Components'. In: *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*. pp. 169–179.