Application of Design for Verification with Concurrency Controllers to Air Traffic Control Software [*]

Aysu Betin-Can Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{aysu,bultan}@cs.ucsb.edu

Mikael Lindvall Benjamin Lux Stefan Topp
Fraunhofer Center for Experimental Software Engineering
4321 Hartwick Road, Suite 500
College Park, MD 20742, USA
{mikli,blux,stopp}@fc-md.umd.edu

ABSTRACT

We present an experimental study which demonstrates that model checking techniques can be effective in finding synchronization errors in safety critical software when they are combined with a *design for verification* approach. We apply the concurrency controller design pattern to the implementation of the synchronization operations in Java programs. This 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 can be verified with respect to arbitrary numbers of threads using infinite state model checking techniques, and the threads which use the controller classes can be checked for interface violations using finite state model checking techniques. We present techniques for thread isolation which enables us to analyze each thread in the program separately during interface verification. We conducted an experimental study investigating the effectiveness of the presented design for verification approach on safety critical air traffic control software. In this study, we first reengineered the Tactical Separation Assisted Flight Environment (TSAFE) software using the concurrency controller design pattern. Then, using fault seeding, we created 40 faulty versions of TSAFE and used both infinite and finite state verification techniques for finding the seeded faults. The experimental study demonstrated the effectiveness of the presented modular verification approach and resulted in a classification of faults that can be found using the presented approach.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.4 [Software Engineering] Software/Program Verification – *Model checking, Formal methods*

General Terms: Design, Verification

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

[*]This work is supported by the NSF grant CCR-0341365, the NASA funded High Dependability Computing Project through NASA cooperative agreement NCC2-1968, and the NSF grant CCF-0438933.

1. INTRODUCTION

Scalability of software model checking depends on extracting compact models from programs that hide the details that are not relevant to the properties that are being verified. This typically requires 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, which enables software developers to document the design decisions that can be useful for verification, may improve the scalability and therefore the applicability of model checking techniques significantly.

In this paper we focus on a design for verification approach for concurrent programming in Java with the goal of eliminating synchronization errors from Java programs using model checking techniques. Concurrent programming in Java is error-prone since it requires conditional waits and notifications implemented with multiple locks and multiple condition variables with associated `synchronized`, `wait`, `notify` and `notifyAll` statements. Concurrent programming using these synchronization primitives results in common errors such as nested monitor lockouts, missed or forgotten notifications, slipped conditions, etc. [19].

The design for verification approach investigated in this paper is based on the concurrency controller design pattern proposed in [2]. Concurrency controller classes developed based on this pattern specify synchronization policies for coordinating the interactions among multiple threads. 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.

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) [5] 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) [28]. In our 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. Since we are only interested in the order of calls to the controller methods by each individual thread, and since the only interaction among different threads is through shared objects that are protected using the concurrency controllers, we can verify each thread in isolation. 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 isolat-

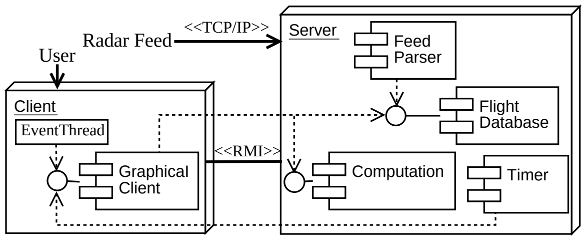


Figure 1: TSAFE architecture.

ing 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 safety critical air traffic control software. First, we reengineered the Tactical Separation Assisted Flight Environment (TSAFE) software based on the concurrency controller design pattern. Then, we created 40 new versions of the TSAFE source code by 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 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 to 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.

2. TSAFE

The Automated Airspace Concept being developed by NASA researchers automates the decision-making in air traffic control by giving the responsibility of determining conflict free trajectories for aircraft to a software system [13, 14]. 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 [15]. To avoid a similar fate, the designers of the Automated Airspace Concept at NASA use a *failsafe short term conflict detection component* in their system, which is responsible for detecting conflicts in flight paths of the aircraft within 1 minute from the current time. If a short term conflict is detected, this component takes over the trajectory synthesis function to direct the aircraft to a safe separation. Since the goal of this component is to provide failsafe conflict detection and resolution capability, it has to be highly dependable, even more than the rest of the system [13, 14].

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 [10]. Later on, as a part of the NASA's High Dependability Computing Project, TSAFE software was integrated into an experimental environment at the Fraunhofer Center for Experimental Software Engineering, Maryland [20]. 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.

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

The TSAFE III implementation consists of 21,057 lines of Java source code in 87 classes. Figure 1 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

The flight database in TSAFE is accessed by multiple threads which may cause failures in its functionality if the threads are not properly synchronized. For example, while the thread running the feed parser is updating the trajectory database, a thread serving an RMI call from a client may be reading it. If such an interaction occurs, an aircraft's location may be displayed incorrectly on the client GUI. Since the client GUI provides the interaction between the human air traffic controllers and the TSAFE system, displaying incorrect information on it could have disastrous effects. To prevent such synchronization problems in Java programs, Java programmers declare the methods of such classes to be synchronized. However, this is not an efficient solution for this case. If the methods of the database are synchronized, then at any given time at most one client thread can access the database. Since client threads never update the database, this synchronization is unnecessary and may slow down the GUI displays. A more appropriate synchronization policy for such cases is to use a 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 our approach, concurrent programmers use the concurrency controller design pattern [2] 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.

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, and 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 2. 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 com-

```
class RWController implements RWInterface {
 int nR; boolean busy; ...
 w_enter = new GuardedCommand() {
  public boolean guard() { return (nR == 0 && !busy); }
  public void update() { busy = true; } };
 w_exit = new GuardedCommand() {
  public boolean guard() { return true; }
  public void update() { busy = false; } };
 r_enter = new GuardedCommand() {
  public boolean guard() { return (!busy); }
  public void update() { nR = nR+1; } };
 r_exit = new GuardedCommand() {
  public boolean guard() { return true; }
  public void update() { nR = nR-1; } };
  ...
}
```

Figure 2: Reader-Writer controller implementation.

mand 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).

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. 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`.

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. To implement a controller, the developer only writes the guarded commands of the actions as shown above and declares the actions as blocking or nonblocking.

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 each 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 state is a state where the corresponding action can be executed, and then updates the current 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 3. 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.

The controller interface is also used to specify when the methods of the shared data objects can be executed. For example, for the

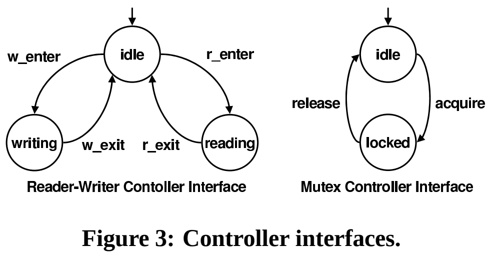


Figure 3: Controller interfaces.

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` 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.

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

Figure 4 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.

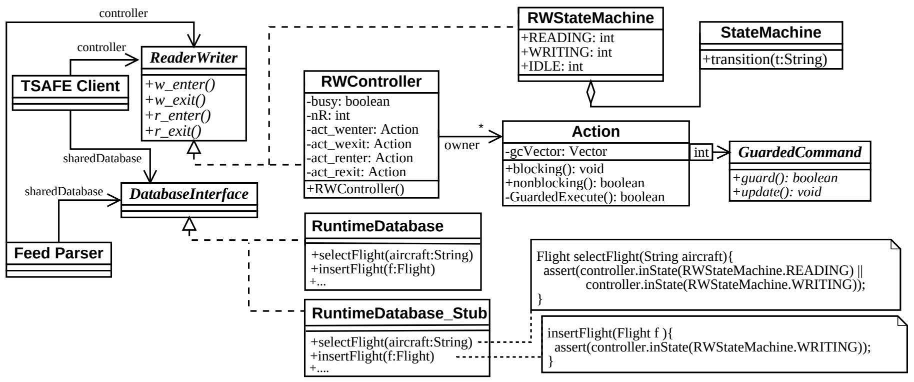
The `RuntimeDatabase` is the implementation of the flight database in TSAFE. The methods of the `RuntimeDatabase` class were synchronized in the original version. Figure 4 shows two of these methods: `insertFlight` which updates the database by inserting a flight, and `selectFlight` which is used to 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`.

The `Action` class in Figure 4 is the helper class which implements the semantics of the action execution. This class is provided with the concurrency controller pattern, i.e., the developers do not need to modify it. Same holds for the `GuardedCommand` Java interface and the `StateMachine` class.

One concern in using the concurrency controller pattern could be the efficiency of the resulting synchronization. We automatically optimize the concurrency controllers using a source-to-source transformation [2]. The optimized controller class 1) uses the specific notification pattern [6], 2) does not have any inner classes, and 3) minimizes the number of method invocations.

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: 1) *Behavior verification:* Verification of the properties of the controller classes assuming that the user threads adhere to 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.



```
Flight selectFlight(String aircraft){
  assert(controller.inState(RWStateMachine.READING) ||
          controller.inState(RWStateMachine.WRITING));
}
```

```
insertFlight(Flight f ){
  assert(controller.inState(RWStateMachine.WRITING));
}
```

Figure 4: Synchronization of the Flight Database in TSAFE using the Concurrency Controller Pattern.

We use the Action Language Verifier (ALV) [5] for behavior verification. We automatically translate the concurrency controllers written based on the concurrency controller pattern into Action Language [2]. ALV supports integer, boolean and enumerated types. This means that we have to restrict the controller variables to these types to verify them with ALV (we use static integers as enumerated variables in the controller implementations). 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.

ALV is an infinite state model checker and can verify specifications with unbounded integer variables (such as nR). This also enables us to use an automated abstraction technique, called counting abstraction, to verify the concurrency controllers with respect to arbitrary numbers of threads [29]. 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. For example the w_enter action in the RW controller is represented in the Action Language as follows:

```
w_enter: pc=IDLE and nR=0 and !busy and busy'=true
                and pc'=WRITING;
```

where the primed variables denote the next-state values. The enumerated variable (pc) keeps track of the thread state which is represented by a state of the controller interface (this is the only information we need to know about a thread to verify the controller implementation). In the parameterized specification generated by counting abstraction, the same action is represented as follows:

```
w_enter: IDLE>0 and nR=0 and !busy and busy'=true
               and WRITING'=WRITING+1 and IDLE'=IDLE-1;
```

Note that, the local variable which 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 can be defined using linear arithmetic constraints on these new variables [29]. 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).

Controller Properties: In order to verify the controllers with ALV we need a list of properties to specify the correct behavior of the

Table 1: RW **Controller Properties**

P1	$AG(busy \Rightarrow nR = 0)$
P2	$AG(busy \Rightarrow AF(\neg busy))$
P3	$AG(\neg busy \wedge nR = 0 \Rightarrow AF(busy \vee nR > 0))$
P4	$\forall x\, AG(nR = x \wedge nR > 0 \Rightarrow AF(nR \neq x))$
P5	$AG(pc = WRITING \Rightarrow AF(pc = IDLE))$
P6	$AG(\neg(pc1 = READING \wedge pc2 = WRITING))$
P7	$EF(pc1 = READING \wedge pc2 = READING)$
P8	$AG(\neg(pc1 = WRITING \wedge pc2 = WRITING))$
P9	$AG(pc1 = READING \Rightarrow nR > 0)$
P10	$AG(pc1 = WRITING \Rightarrow busy)$
P11	$AG(WRITING > 0 \Rightarrow AF(WRITING = 0))$
P12	$AG(\neg(READING > 0 \wedge WRITING > 0))$
P13	$EF(READING \geq 2)$
P14	$AG(\neg(WRITING > 1))$
P15	$AG(READING = nR)$
P16	$AG(WRITING = 1 \Leftrightarrow busy)$
P17	$\forall x\, AG(READING = x \wedge READING > 0$ $\Rightarrow AF(READING \neq x))$

controllers, i.e., we need the class invariants of the controller classes. We allow the CTL properties for the controllers to be either inserted directly to the generated Action Language specification or written as annotations in the controller classes (which are then automatically inserted into the Action Language translation).

The properties for the RW controller are shown in Table 1. The properties P1–4 only refer to the variables of the controller class. For example, the global property P1 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 abstract Action Language specifications. The properties P5–10 are for concrete specifications and refer to concrete thread states. For example the property P5 states that whenever a thread is in the WRITING state it will eventually reach the IDLE state. The properties P11–17 are for the parameterized instances and refer to the integer variables which represent the number of threads in a particular state. For example property P15 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 1 contain universally quantified integer variables. We are able to check such properties using ALV by declaring the universally quantified variables as parameterized constants.

5. INTERFACE VERIFICATION AND THREAD ISOLATION

During interface verification we verify each thread in the program separately using the program checker Java Path Finder (JPF) [28]. A thread is correct if it adheres to the interfaces of the controllers and accesses the shared data protected by the controllers at the allowed interface states (as specified in the data stubs). If a thread calls the actions of a controller in a sequence that is not defined by the interface of that controller, then the thread does not obey the controller interface. As explained in Section 3, the methods of the controller interfaces and data stubs have assertions which check the above criteria. If JPF reports the violation of an assertion in a controller interface or a data stub, then we know that the thread that is being verified is not correct. JPF outputs a counter-example execution trace that leads to the violation of the assertion.

Here, we will introduce a simple model for distributed programs in order to explain our interface verification technique. 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}). Our interface verification approach is thread modular, i.e., we check each thread separately for interface violations. To this end, we isolate each thread T_i by a conservative approximation of the behavior of other threads in the distributed program without modifying the code of T_i.

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 its program counter. The state of a thread T is represented with the shared store and 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. To simplify the discussion, we assume that these operations are performed by method calls, which is a reasonable assumption in object oriented programming. 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 the remote programs. (We name the outgoing RMI call an RMI operation and the incoming RMI call an RMI event.) We isolate a thread interaction via interaction operations with a transformation function $\mathcal{F}$. Below we define this function for the interaction operations. Then, we discuss modeling input events with drivers.

Modeling Interaction Operations with Stubs: In a (distributed) program implemented based on the concurrency controller pattern the elements of shared store are implemented explicitly. These elements, which are controllers and shared data protected by these controllers, are known at the verification phase. We isolate the interaction through a controller as follows. Let C denote a concurrency controller and let $CI = (Q, q_0, A, R)$ be the interface of C, where Q is the set of interface states, $q_0 \in Q$ is the initial state, A is the set of the controller actions, and $R \subseteq Q \times A \times Q$ is the transition relation. Although we handle nondeterministic interfaces, here, to simplify the discussion, we assume that there is at most one next state for each state and action pair. We replace the controller C in the program with CI, and at run-time we create one CI instance per thread. Note that, although the instances of C are shared, the instances of CI are not shared. Let T be a thread that accesses controller C. Whenever the control state of T is at a controller action act of C, T updates its current interface state ($\mathtt{cur} \in Q$) according to the transition relation R. If such transitions are not defined (i.e., $\nexists q' \in Q, (\mathtt{cur}, act, q') \in R$) then T does not obey the controller interface CI and an interface violation has occurred. Formally, for each action $act \in A$ we apply the transformation function $\mathcal{F}(C.act()\{body\} : r) = CI.act()\{$ `assert` $(\exists q' \in Q, (\mathtt{cur}, act, q') \in R) \wedge$ `cur` $:= t' \wedge (\mathtt{cur}, act, t') \in R\} : r$ where $body$ denotes the method body of act and r is the return type of act. With this transformation, the internals of the controller action (conditional waits, guarded commands, etc.) are abstracted with the action sequencing rules the thread has to obey. Controller interfaces are in the local store of the thread and any interface operation only influences the $Local$ of the thread. Therefore, the controller is removed from the $Shared$.

We isolate the interaction through a shared data as follows. Let $sho \in Shared$ be a shared data object protected by the controller C. Let C be abstracted with the controller interface CI as explained above. Consider the case where the control state of T is at a read or write operation from sho through a method call $m(a_0, a_1, .., a_l)\ \{body\} : r, E$ where $a_0, a_1, .., a_l$ is the sequence of arguments, r is the return type of m and E is the set of exception types that can be thrown by m. Recall that, during interface verification our goal is to check if T invokes m at an allowed interface state of CI. Let Q_m be the set of allowed interface states to invoke method m. The transformation function for this read operation is $\mathcal{F}(m(a_0, a_1, \ldots, a_l)\{body\} : r, E) = m'(a_0, a_1, \ldots, a_l)\{$ `assert` $(\mathtt{cur} \in Q_m) \wedge ($`throw choose`$(E) \vee$ `return choose` (r) `)` $\}\ : r, E$ where `choose` returns a nondeterministically chosen value of its argument type and if its argument is a type set, a type is chosen nondeterministically before returning a value of that type. During the interface verification, each of these nondeterministic choices are explored; hence, the stub methods conservatively approximate the influence of other threads on T. Based on the concurrency controller pattern, the assertion and Q_m are specified by the programmer in the data stub for sho. If this assertion fails, then T has performed an illegal operation and an interface violation has occurred. In our concurrency controller pattern, the stub methods for shared data sho are encapsulated in the shared data stub class of 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, is not sufficient to isolate a thread for some distributed programs such as TSAFE. The steps above only abstract the interaction operations 1 and 2. Here we explain how we model the rest of the interaction operations. An RMI operation is a method call $m_{RMI}(a_1, a_2, .., a_l) : r, E$. When a thread T invokes the remote method m_{RMI}, the thread is affected only by the return value and the exceptions thrown by m_{RMI}. We isolate the thread from this operation by transforming the remote method call $m_{RMI}(a_1, a_2, .., a_l) : r, E$ to a stub method call with the same signature. This stub call returns all possible return values of type r and throws all possible exceptions in E. With this stub method call, we conservatively approximate the return value and exceptions thrown. To apply this transformation, the remote methods have to be identified statically. In an object oriented language, a remote operation is performed by $remote.m_{RMI}$ $(a_1,\ a_2,\ ..,a_l)$ where $remote$ denotes the remote object. Therefore, we first identify the remote objects and their methods, and then, we substitute each remote object $remote$ with a remote stub $remote'$ which includes a stub method for each remote method of $remote$.

Modeling the rest of the interaction operations follows the same principle. 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. Formally, $\mathcal{F}(m(a_0, \ldots, a_l)\{body\} : r, E) = m'(a_0, ..., a_l)\{$ `throw choose`(E) $\vee$ `return choose`(r) $\} : r, E$ where m is an operation of type 4–7. Unlike the remote methods, the interaction methods for 4–7 are predetermined. For example, in Java, the graphical methods are within libraries such as the `java.awt` or `javax.swing` library. Therefore, we use one-time-implemented stub methods while transforming such operations. (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 transformation function. We implemented `choose` with the JPF's nondeterminism utilities `Verify.random(int)` and `Verify.randomBool()`. These utilities force JPF to search exhaustively 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.

Modeling Thread Initialization and Input Events with Drivers: The counterpart of stubs in thread isolation are the drivers. Drivers are necessary to transform each thread execution to 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 similarly. In other words, these drivers set the initial configuration for $Shared$ and $Local$.

In TSAFE, the implicitly created threads are the event thread that dispatches the GUI events and the threads created to serve RMI calls. An RMI thread T_{RMI} is responsible for serving the incoming RMI calls, i.e. the execution of T_{RMI} is a sequence of remote events. We automatically generate the driver of T_{RMI} that overapproximates such 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.

The Java code for the drivers of these threads is generated automatically. They consist of a single 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 an infinite loop. 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.

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 correctness conditions discussed above. Using the results of this data dependency analysis user has to restrict the input domains.

Data Dependency Analysis: It is possible that some of the input parameters (or the return values) that are passed to a thread via drivers and stubs may not influence the synchronization behavior of a thread. 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 [22]. 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 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 traversal should be interprocedural and capture the implicit dependencies between the methods of the same class such as the dependency between a get method and a set method of the same class. The result of this procedure is a backward dependence tree per starting point whose vertices are the collected definition sites. The leaves are the influencing argument sequence elements and a path in the tree shows how these elements control the execution of the shared operations.

We implemented this analysis using the Soot Java optimization framework [24], which uses a 3-address representation for Java. The analysis determines which statements, directly or indirectly, affect the reachability of invocation of a method that belongs to a concurrency controller or a shared data class. The analysis we implemented is context insensitive. In the implementation, instead of computing the program dependence graph, the control and data dependencies are computed on the fly. To capture the implicit dependencies, before the traversals, for each class field we compute the set of methods updating its value (and the value of its elements).

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 is given in the drivers or the stubs. For the ones that might influence the synchronization behavior there are two possibilities. If the domain of such a value is finite (e.g. boolean) we enumerate all possible values and choose one value using JPF's nondeterminism utilities. Otherwise, the analysis results are inspected and necessary values are provided by the user. The value is chosen from this predetermined value set with the `Verify.random(int)` utility of JPF.

We have tried to use the slicers available in the Bandera [11] and Indus [18] 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 dependencies above.

5.1 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. When a Java program has GUI objects, a thread (called `EventDispatchThread`) is instantiated implicitly. The RMI thread is implicitly created when there is an RMI call from the server component to the client 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. 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 isolate the event thread with a driver and the GUI stubs provided by our framework. The driver generation is semi-automated. We automatically generate an event thread driver and expect the user to perform data value assignments using the results of the data dependency analysis explained above.

The automatically generated driver first launches the GUI components, and finds all the visible and enabled GUI objects that have registered event listeners. Then, it enters a loop generating the input event sequence. In each iteration of the loop, the driver first chooses one of these GUI objects and then chooses an event and calls the listeners for that event.

During driver generation 1) the GUI component launch mechanism is created by copying the relevant part from the application code, for example, in this case, from the TSAFE client main program, and 2) all possible user event types are identified by finding all the different event listener types in the code.

The RMI Thread: The TSAFE's client component has 2 RMI operations and 4 RMI events. We have implemented a generator that inspects a remote interface, which is a Java interface that extends `java.rmi.Remote`, and creates one stub to model the RMI operations and one driver to model the RMI thread. The generator inspects the remote interface in the server component to collect the RMI operations to synthesize the RMI stub. Similarly, the generator inspects the remote interface in the client component to collect the RMI events to synthesize the RMI driver.

A driver is also responsible for initialization besides producing all possible input event sequences. Therefore, the generator examines 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 generator creates the code for registering the RMI stub of corresponding component to the `Naming` class. Then the generator puts an instantiation of the given concrete class into the code. After the generation of the driver, the user can modify the parameter value assignments depending on the results of the dependency analysis discussed above.

As a result, the generated RMI driver for the TSAFE client first registers an RMI stub of the TSAFE server component, and instantiates the client component. Then it produces all possible event sequences with these 4 input events.

5.2 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 event thread driver first finds the `Timer` object, and then calls its registered listener in an infinite loop.

To isolate the RMI thread at the server component we have applied the techniques discussed above. 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 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 substitution discussed above. In the next section 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 initialize 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. 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. 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 only the characters denoting the message type and the exceptions 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. During this 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.

First, 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. 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).

Table 2: 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

6.1 Fault Seeding

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

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

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

After the fault seeding, the FC-MD team sent the modified versions back to the UCSB team. Table 2 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.

6.2 Results and Discussion

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

As shown in Table 2, there were a total of 14 controller faults and 26 interface faults in versions v1–40. When we verified the controllers in versions v1–40 with ALV we found 12 faults in the controllers. The faults that were not found by ALV were the faults in versions v5 and v13 which were spurious faults, i.e., they are modifications in the controller classes which do not cause any failures in the controller behavior. For example, the modification in v13 changed an assignment in the `w_exit` action of the RW controller from `busy=false` to `busy=!busy`. However, this modification does not cause any failures since `busy` is always `true` when `w_exit` is called. The modification in v5 changed the `release` action in the MUTEX controller from blocking to nonblocking. Again this modification does not change the behavior of the controller since the guard of the `release` action is `true`, i.e., it never blocks.

Among the 26 interface faults, interface verification using JPF identified 21 of them. Two of the faults (v22 and v33) that were not caught by JPF were spurious faults. However, the faults in versions v18, v19, and v20 were real faults which can cause failures but were not found by JPF. We will discuss these faults in detail below.

Table 3 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 two columns show the memory and time consumed for the verification of an instance without any faults and the last two columns show the average memory and time consumed for counter-example generation for the instances with faults.

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 the table). 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 (P1–10 in Table 1) and 11 properties on the parameterized instance (P1–4 and P11–17 in Table 1). Both verification and falsification of the MUTEX controller is more efficient compared to RW controller since it is a smaller specification with less number of variables.

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 arbitrary number of threads. However, for falsification the results of the concrete and parameterized instances are equivalent, both of them generate 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 arbitrary number of threads.

JPF Performance: The second part of Table 3 shows the performance of JPF for interface verification of each thread as explained in 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 used in the environment. Still, overall, falsification performance is better than the verification performance, especially for the more challenging verification tasks such as the Client-Event thread and the Server-Feed thread.

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

Table 3: Verification and falsification performance

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

Component-Thread	Verify		Falsify	
	M(MB)	T(sec)	M(MB)	T(sec)
Client-Main	2.32	2.00	–	–
Client-Event	33.09	663.21	12.2	15.63
Client-RMI	40.96	17.06	42.64	10.12
Server-Main	67.72	17.08	–	–
Server-Event	10.95	6.57	9.56	6.88
Server-RMI	20.31	91.79	24.74	29.43
Server-Feed	83.49	123.12	94.72	18.51

an error. However, such errors can be avoided by using the automated optimization step since that step only uses the initialization statements, guards, updates and blocking/nonblocking declarations, i.e., the parts verified during behavior verification.

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, 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 [4, 18] 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 three real faults that were missed by the presented verification approach: the interface faults in versions v18, v19, and v20. The versions v17, v18, v19, and v20 were all created by adding a branch condition in front of a method call to a controller. The added branch condition tests if the value of a variable is less than a constant. If not, the call to the controller method is skipped. The variable in the branch condition is initialized to zero and is incremented every time the control reaches the inserted branch condition. The only difference between the faults in these versions was the constant value in the branch conditions which was 100, 1000, 10000, and 100000, for versions v17, v18, v19, and v20, respectively. Interface verification with JPF identifies the fault in v17 however misses the faults in v18, v19, v20. Clearly, these are convoluted faults. This fault type was suggested by the UCSB team as a way to challenge the interface verification step. These faults demonstrate that there is a limit to the depth of the faults that can be identified using explicit state verification techniques without running out of memory. In order to deal with this type of faults symbolic analysis of the branch conditions may be necessary.

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

7. RELATED WORK

This paper builds on our earlier work on the concurrency controller pattern discussed in [2]. Also, in [3], we presented a related design pattern, called the peer controller pattern, for design and verification of asynchronously communicating web services. The work in [3] 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 the current paper are: 1) An experimental study demonstrating the applicability of the design for verification approach to safety critical air traffic control software and empirical results demonstrating the effectiveness of our modular verification strategy. 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.

There have been other studies on design for verification. The approach in [23] 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 [21]. However, our interface-based modular verification technique is different than the approach in [21].

In [8] interfaces of software modules are specified as a set of constraints, and algorithms for interface compatibility checking are presented. In [9] 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 [1, 7, 11]. 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 [17] 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 [26, 27] 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 [25] 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 [11] 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 [12] is similar to out 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 [16] 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.

8. CONCLUSIONS

We presented a design for verification approach for eliminating synchronization errors in Java programs based on a design pattern for writing synchronization policies. The concurrency controller pattern supports a modular verification strategy by identifying the stateful interfaces of concurrency controllers. Based on these interfaces, verification of the synchronization policies implemented as concurrency controllers can be separated from the verification of their correct usage by different threads. We presented techniques for thread isolation which enables verification of each thread separately. To investigate the effectiveness of this design for verification approach on safety critical air traffic control software we reengineered the TSAFE software using the concurrency controller design pattern and created modified versions of the reengineered TSAFE code using fault seeding. The presented verification techniques were able to find almost all of the seeded faults. The experimental study resulted in a fault classification and helped us identify new directions for improving the presented approach.

9. REFERENCES

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of SPIN Workshop*, pages 103–122, 2001.

[2] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proc. of ASE*, pages 248–257, 2004.

[3] A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In *Proc. of the 14th Int. World Wide Web Conf.*, pages 750–759, 2005.

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

[5] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proc. of ASE*, pages 382–386, 2001.

[6] T. Cargill. Specific notification for Java thread synchronization. In *Proc. of the 3rd PLoP*, 1996.

[7] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. of ICSE*, pages 385–395, 2003.

[8] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdziński, and F. Mang. Interface compatibility checking for software modules. In *Proc. of CAV*, pages 428–441, 2002.

[9] R. DeLine and M. Fahndrich. Typestates for objects. In *Proc. of ECOOP*, pages 465–490, 2004.

[10] G. Dennis. TSAFE: Building a trusted computing base for air traffic control software, Master's Thesis, 2003.

[11] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proc. of ICSE*, pages 177–187, 2001.

[12] M. B. Dwyer, Robby, O. Tkachuk, and W. Visser. Analyzing interaction orderings with model checking. In *Proc. of ASE*, pages 154–163, 2004.

[13] H. Erzberger. The automated airspace concept. In *Proc. of USA/Europe Air Traffic Management R&D Seminar*, 2001.

[14] H. Erzberger. Transforming the NAS: The next generation air traffic control system. In *Proc. of the 24th Int. Congress of the Aeronautical Sciences*, 2004.

[15] Advance automation system. Dep. of Transportation, Office of Inspector General, Audit Report, AV-1998-113, 1998.

[16] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proc. of the SPIN Workshop*, pages 213–224, 2003.

[17] P. Godefroid, C. Colby, and L. Jagadeesan. Automatically closing open reactive programs. In *Proc. of PLDI*, pages 345–357, 1998.

[18] Indus. http://indus.projects.cis.ksu.edu.

[19] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts, 1999.

[20] M. Lindvall and et al. An evolutionary testbed for software technology evaluation. *NASA Journal of Innovations in Systems and Software Engineering*, 1(1):3–11, 2005.

[21] P. C. Mehlitz and J. Penix. Design for verification using design patterns to build reliable systems. In *Workshop on Component-Based Soft. Eng.*, 2003.

[22] J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *ACM Software Engineering Notes*, pages 177–184, 1984.

[23] N. Sharygina, J. C. Browne, and R. P. Kurshan. A formal object-oriented analysis for software reliability: Design for verification. In *Proc. of FASE*, pages 318–332, 2001.

[24] Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/.

[25] S. D. Stoller and Y. A. Liu. Transformations for model checking distributed java programs. In *Proc. of the SPIN Workshop*, pages 192–199, 2001.

[26] O. Tkachuk and M. B. Dwyer. Adapting side-effects analysis for modular program model checking. In *Proc. of ASE*, pages 188–197, 2003.

[27] O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proc. of ESEC/FSE*, pages 116–129, 2003.

[28] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[29] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proc. of ISSTA*, pages 169–179, 2002.