# Assume-guarantee Verification of Source Code with Design-Level Assumptions

Dimitra Giannakopoulou
*RIACS/USRA*
*NASA Ames Research Center*
*Moffett Field, CA 94035-1000, USA*
*dimitra, pcorina@email.arc.nasa.gov*

Corina S. Păsăreanu
*Kestrel Technology LLC*

Jamieson M. Cobleigh*
*Department of Computer Science*
*University of Massachusetts*
*Amherst, MA 01003-9264, USA*
*jcobleig@cs.umass.edu*

## Abstract

*Model checking is an automated technique that can be used to determine whether a system satisfies certain required properties. To address the "state explosion" problem associated with this technique, we propose to integrate assume-guarantee verification at different phases of system development. During design, developers build abstract behavioral models of the system components and use them to establish key properties of the system. To increase the scalability of model checking at this level, we have developed techniques that automatically decompose the verification task by generating component assumptions for the properties to hold. The design-level artifacts are subsequently used to guide the implementation of the system, but also to enable more efficient reasoning at the source code-level. In particular, we propose to use design-level assumptions to similarly decompose the verification of the actual system implementation. We demonstrate our approach on a significant NASA application, where design-level models were used to identify and correct a safety property violation, and design-level assumptions allowed us to check successfully that the property was preserved by the implementation.*

## 1. Introduction

Our work is motivated by an ongoing project at NASA Ames Research Center on the application of automated verification techniques to autonomous software. Autonomous systems involve complex concurrent behaviors for reacting to unpredicted environmental stimuli without human intervention. Extensive verification is a pre-requisite for the deployment of missions that involve autonomy.

Given some formal description of a system and of a required property, model checking automatically determines whether the property is satisfied by the system. The limitation of the approach, commonly known as the "state-explosion" problem, is the exponential relation of the number of states in the system under analysis to the number of components of which the state is made [28]. Model checking therefore does not scale, in general, to systems of realistic size.

The aim of our work is to increase the applicability and scalability of model checking by:

1. applying it at different phases of software development, and
2. using compositional (i.e. divide and conquer) verification techniques that decompose the verification of a software system into manageable subparts.

We believe that the verification of a safety critical system should be addressed as early as during its design, and should should go hand in hand with later phases of software development. Our experience working closely with the developers of the control software for an experimental Mars Rover has been that several integration issues can be detected during system design. At that level, verification of the system is typically more manageable and errors are easier and cheaper to fix since the system has not yet been implemented. Although system verification at the design level is undoubtedly important, there is little guarantee that the implemented system indeed satisfies the properties established at design time. We therefore need to provide a means of establishing that system implementations preserve the properties that have been demonstrated at the design level.

During design, our work supports the verification of Labeled Transition Systems (LTSs) against safety properties expressed in terms of finite-state automata. LTSs are communicating finite-state machines that can be used to describe the behavioral interfaces of software or hardware components. Safety properties describe the legal (and illegal) sequences of actions that a system can perform.

In previous work, we developed novel techniques for performing automated assume-guarantee verification at the

*This author is grateful for the support received from RIACS to undertake this research while participating in the Summer Student Research Program at the NASA Ames Research Center.

design level [9, 16]. Assume-guarantee reasoning was originally aimed at enabling the stepwise development of concurrent processes, but has more recently been used to decompose the verification of large and complex systems. It is in the latter context that we use it in our work.

Assume-guarantee reasoning first checks whether a component $M$ guarantees a property $P$, when it is part of a system that satisfies an assumption $A$. Intuitively, $A$ characterizes all contexts in which the component is expected to operate correctly. To complete the proof, it must also be shown that the remaining components in the system ($M$'s environment), satisfy $A$. In contrast with previous assume-guarantee frameworks [8, 19, 25, 30], our techniques do not require human input in defining assumptions, but rather generate assumptions *automatically*, thereby increasing the accessibility of this kind of reasoning.

The focus of the present work is to develop a methodology for using design artifacts to leverage the verification of the actual system implementation. To this aim, we propose to use the assumptions that are automatically generated during design-level verification to perform assume-guarantee reasoning at the implementation level. In general, we believe that design-level assumptions can be used both during component development as an adjunct to traditional unit testing approaches, and during program validation, to enable more efficient reasoning and to model non-software components, including the actual environment of a reactive system. For the latter, it may be the case that critical system properties can only be demonstrated under specific environmental assumptions that appear reasonable to the developer, but cannot be discharged because the environment is unknown (e.g., autonomous systems). These assumptions can then be used to monitor, during deployment, the behavior of the environment, and trigger recovery actions when this behavior falls outside the envelope defined by the assumption.

The work presented in this paper contributes:

1. a methodology for using the results of the modular analysis at the design level to improve the performance of verification tools at the code level;
2. a program instrumentation technique for supporting assume-guarantee reasoning of Java programs using the Java PathFinder (JPF) model checker developed at NASA Ames [32]; and
3. a significant case study demonstrating the applicability of our approach to a real NASA software system.

The case study has been performed in the context of an ongoing collaboration with the developers of the control software for an experimental Mars Rover. More specifically, we have used our techniques to verify several versions of the software both during its design, and during its implementation, often using the results of our work to influence the design decisions of the developers. In this paper we will present how design-level models were used to identify and correct a safety property violation, and how design-level assumptions allowed us to check successfully that the property was preserved by the system implementation.

Note that, even though our research to-date has focused on checking implementations using software model-checking tools, we are aware of the fact that for complex software, even components may be too complicated to analyze exhaustively. In such cases, we intend to sacrifice exhaustiveness for the sake of scalability by using lighter-weight analysis techniques such as stateless model checking [17] or runtime analysis [20].

The remainder of the paper is organized as follows. We first provide some background on our design-level verification techniques in Section 2, followed by a description of the methodology that we propose in Section 3. Section 4 presents our approach to model checking source code in an assume-guarantee style. Section 5 describes the experience and results obtained by the application of our methodology to a NASA system that was the focus of our case study. Finally, Section 6 presents related work and Section 7 concludes the paper.

## 2. Background: Assume-guarantee verification at the design level

In this section we give background on assume-guarantee reasoning and we describe the automated assume-guarantee frameworks that we have developed for reasoning about software systems at the design level.

### 2.1. Assume-guarantee reasoning

In the assume-guarantee paradigm a formula is a triple $\langle A \rangle\ M\ \langle P \rangle$, where $M$ is a component, $P$ is a property, and $A$ is an assumption about $M$'s environment. The formula is true if whenever $M$ is part of a system satisfying $A$, then the system must also guarantee $P$.

Consider for simplicity a system that is made up of components $M_1$ and $M_2$. To check that the system satisfies a property $P$ without composing $M_1$ with $M_2$, one can apply assume-guarantee reasoning as follows. The simplest assume-guarantee proof rule shows that if $\langle A \rangle\ M_1\ \langle P \rangle$ and $\langle true \rangle\ M_2\ \langle A \rangle$ hold, then $\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle$ is true. This proof strategy can also be expressed as an inference rule:

$$\frac{\begin{array}{ll}\text{(Premise 1)} & \langle A \rangle\ M_1\ \langle P \rangle \\ \text{(Premise 2)} & \langle true \rangle\ M_2\ \langle A \rangle\end{array}}{\langle true \rangle\ M_1 \parallel M_2\ \langle P \rangle}$$

Note that for the use of this rule to be justified, the assumption must be more abstract than $M_2$, but still reflect $M_2$'s behavior. Additionally, an appropriate assumption for the rule needs to be strong enough for $M_1$ to satisfy $P$.

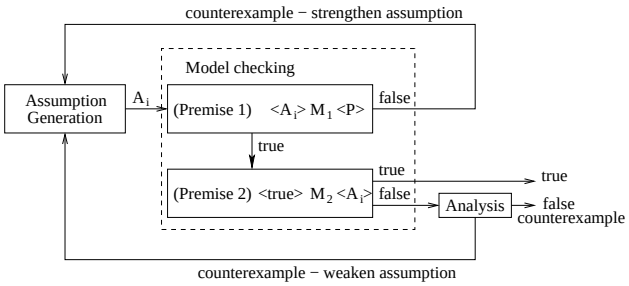


**Figure 1. Iterative framework for assume-guarantee reasoning**

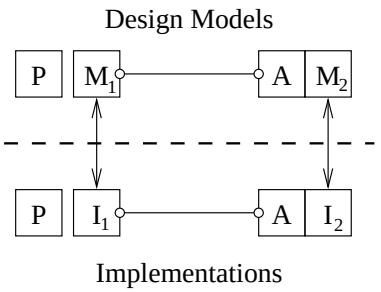


**Figure 2. Verification at design and code level**

### 2.2. Automated assume-guarantee frameworks

Several frameworks have been proposed [8, 19, 25, 30] to support assume-guarantee reasoning. However, their practical impact has been limited because they require non-trivial human input in defining assumptions. In previous work [9, 16] we developed novel frameworks to perform assume-guarantee reasoning in a *fully automatic* fashion. The work was done in the context of *finite* labeled transition systems with blocking communication and *safety* properties.

In [16], we present an approach to synthesizing the assumption that a component needs to make about its environment for a given property to hold. The assumption produced is the *weakest*, that is, it restricts the environment no more and no less than is necessary for the component to satisfy the property. The automatic generation of weakest assumptions has direct application to the assume-guarantee proof; it removes the burden of specifying assumptions manually thus automating this type of reasoning.

The algorithm presented in [16] does not compute partial results, meaning no assumption is obtained if the computation runs out of memory, which may happen if the state-space of the component is too large. We address this problem in [9], where we present a novel framework for performing assume-guarantee reasoning using the above rule in an incremental and fully automatic fashion. This framework is illustated in Figure 1.

At each iteration, a learning algorithm is used to build approximate assumptions $A_i$, based on *querying* the system and on the results of the previous iteration. The two premises of the compositional rule are then checked. Premise 1 is checked to determine whether $M_1$ guarantees $P$ in environments that satisfy $A_i$. If the result is false, it means that this assumption is too *weak*, and therefore needs to be *strengthened* with the help of the counterexample produced by checking premise 1. If premise 1 holds, premise 2 is checked to discharge $A_i$ on $M_2$. If premise 2 holds, then the compositional rule guarantees that $P$ holds in $M_1 \parallel M_2$. If it doesn't hold, further analysis is required to identify whether $P$ is indeed violated in $M_1 \parallel M_2$ or whether $A_i$ is stronger than necessary, in which case it needs to be *weakened*. The new assumption may of course be too weak, and therefore the entire process must be repeated. This process is guaranteed to terminate; in fact, it converges to an assumption that is necessary and sufficient for the property to hold in the specific system.

Recently, we have extended our frameworks to handle circular rules and more than two components. We have implemented the frameworks in the LTSA model-checking tool [26] and have applied them to the verification of several design models of NASA software systems.

## 3. Methodology

This section describes our methodology for using the artifacts of the design-level analysis in order to decompose the verification of the implementation of a software system.

To address the scalability issues associated with software model checking, our approach, illustrated in Figure 2, integrates assume-guarantee reasoning of concurrent systems at the design and at the implementation level. At the design level, the architecture of a system is described in terms of components and their behavioral interfaces modeled as LTSs. Design level models are intended to capture the design intentions of system developers, and allow early verification of key integration properties.

For example, consider a system that consists of two design level components $M_1$ and $M_2$, and a global safety property $P$ (describing the sequence of events that the system is allowed to produce, or equivalently the bad behaviors that the system must avoid). To check in a more scalable way that the composition $M_1 \parallel M_2$ satisfies $P$, we use the assume-guarantee frameworks described in the previous section. We expect that, with the feedback obtained by our verification tools, the developers of the system will correct their models until the property is achieved at the design level. At that stage, our frameworks will have automatically generated an assumption $A$ that is strong enough for $M_1$ to satisfy $P$ but weak enough to be discharged by $M_2$ (i.e. $\langle A \rangle\ M_1\ \langle P \rangle$ and $\langle true \rangle\ M_2\ \langle A \rangle$ both hold).

To then establish that the property is preserved by the actual implementation, our approach uses the automatically generated assumption $A$, to perform assume-guarantee reasoning at the source code level. The implementation is decomposed as specified by the architecture at the design level (i.e. components $I_1$ and $I_2$ implementing $M_1$ and $M_2$, respectively; see Figure 2), and we establish that $\langle true \rangle\ I_1 \parallel I_2\ \langle P \rangle$ holds by checking that $\langle A \rangle\ I_1\ \langle P \rangle$ and $\langle true \rangle\ I_2\ \langle A \rangle$. If the two premises are true then the correctness of the assume-guarantee rule guarantees that the property is preserved by the implementation. If any one of the two premises does not hold, then the counterexample(s) obtained expose some incompatibility between the models and the implementations, and are used to guide the developers in correcting the implementation, the model, or both.

Alternatively, one may wish to check preservation of properties by checking directly that each implemented component refines its model. In our experience, for well designed systems, the interfaces between components are small, and the assumptions that we generate are much smaller than the component models. Moreover, the controllability information that we use to derive these assumptions, and the fact that we take the properties into account in building them, typically allow us to achieve further reduction than abstraction techniques that would merely simplify models based on component interfaces [16]

The software architecture of a system may not always provide the best decomposition for verification [7]. However, we currently focus on this line of research because one of our target applications is the Mission Data Systems architecture (MDS) [12].MDS allows adaptations to be constructed by configuring instantiations of components with an ADL. We are interested in enriching critical components of the MDS system with models describing their abstract behavioral interfaces, and relating the analysis of these models with analysis of the resulting implementation.

## 4. Assume-guarantee analysis of software

In this section, we describe the main challenges in extending the Java Pathfinder software model checker to perform assume-guarantee reasoning, with assumptions and properties expressed as finite-state machines. Although we make our presentation in the context of Java programs, our approach extends to other programming languages and model checkers.

### 4.1. Java PathFinder

For checking Java implementations, we use Java PathFinder (JPF) [32]. JPF is an explicit-state model checker that analyzes Java bytecode classes directly for deadlocks and assertion violations. JPF is built around a special-purpose Java Virtual Machine (JVM) that allows Java programs to be analyzed. JPF supports depth-first, breadth-first as well as heuristic search strategies to guide the model checker's search in cases where the state-explosion problem is too severe [18].

In addition to the standard language features of Java, JPF uses a special class `Verify` that allows users to annotate their programs so as 1) to express non-deterministic choice with methods `Verify.random(n)` and `Verify.randomBool()` and 2) to truncate the search of the state-space with method `Verify.ignoreIf(condition)`, when the `condition` becomes true. Methods `Verify.beginAtomic()` and `Verify.endAtomic()` respectively indicate the start and end of a block of code that the model cheker should treat as one atomic statement and not interleave its execution with any other threads.

### 4.2. Mapping and instrumentation

We instrument Java programs to perform assume-gurantee reasoning using Java PathFinder. In our framework, both assumptions and properties are expressed as deterministic finite-state machines. For example, consider a program that opens and closes files. The assumption illustrated in Figure 3 expresses the fact that the environment will always open a file before closing it, and will always perform these actions in alternation. Any different behavior with respect to these actions leads the assumption to the `ignore` state, which reflects the fact that such behavior will never be exercised in the context of the environment that the assumption represents. On the other hand, the property illustrated in Figure 4 expresses the fact that the system is *required* to always open a file before closing it, and to always perform these actions in alternation. Any behavior that does not conform to this pattern is violating, and will be trapped in the `error` state.

At the source code level, assumptions and properties will be used to examine the behavior of the system and check whether behaviors that are not ignored by the assumption may be trapped by the property, meaning that the property is violated under the specific assumption. A necessary step in our approach is therefore a mapping between actions that appear in the design-level assumptions and properties, and events that occur in the software. For simplicity, we are assuming that actions in our design models correspond in the software either to method calls or to the locking and unlocking of objects.

The software must then be instrumented so that each event that appears in the mapping gets trapped, and is used in examining its effects on the state of the assumption and property. Presently, this instrumentation is done by hand, but we are considering the use of automated tools such as

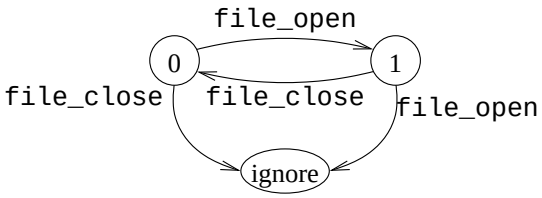


**Figure 3. Example assumption**

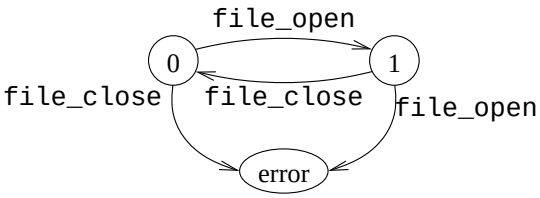


**Figure 4. Example property**

```
    public static void event() {
1)    Verify.beginAtomic()
2)    String threadName = Thread.currentThread().getName();
3)    Throwable throwable = new Throwable();
4)    StackTraceElement st = (throwable.getStackTrace())[1];
5)    String methodName = st.getMethodName();
6)    String className = st.getClassName();
7)    int eventID = getEvent(className, methodName, threadName);
8)    AG_Assumption.event(eventID);
9)    AG_Property.event(eventID);
10)   Verify.endAtomic();
    }
```

**Figure 5. Method `event` of class `AG_Monitor`**

[20]. We have to date experimented with the use of iContract [23], but unfortunately encountered bugs in the software which could not be fixed because the tool is no longer being supported.

Our instrumentation adds at each point where an event occurs, a method call `AG_Monitor.event()`, which traps the event and calls methods of the assumption and the property. This method, shown in Figure 5, uses Java reflection to determine the name of the thread making the method call (line 2), the method being called (lines 3-5), and the class that contains the method (line 6). These three pieces of information are used as a key to look up the corresponding event from the design level model (line 7). Then, this event is passed on to the assumption (line 8) and to the property (line 9). The entire block is enclosed by JPF directives (lines 1 and 10) which instruct it to treat the method body as an atomic step and to interleave no other threads with the execution of this method.

If more information is needed to determine the mapping between the Java program and the events from the design-level model, then the `event` method can be extended to allow parameters to be passed that contain this extra information. This was necessary in our case study to obtain information about parameters being passed into method calls, parameters being returned from method calls, and to trap locks and unlocks of objects.

Properties and assumptions are implemented by classes `AG_Assumption` and `AG_Property`. An excerpt of the `AG_Assumption` class is shown in Figure 6. This class has a static integer field that records the current state of the assumption automaton (line 1) and a transition table that

```
   public class AG_Assumption {
1)   private static int state = 0;
2)   private static int[][] trans;
     ...
     public static void event(int e) {
3)     state = trans[state][e];
4)     Verify.ignoreIf(state < 0);
     } }
```

**Figure 6. Class `AG_Assumption` (excerpt)**

stores the transitions (line 2). The `ignore` state of the assumption is represented by a state with an ID less than zero. The method `event` advances the assumption by looking up the next state in the transition table (line 3). If the state is less than 0, this represents that the current execution does not satisfy the assumption and that JPF should not continue exploring this path (line 4). The current path does not need to be further explored, since we are only interested in property violations that occur under the given assumption.

The `AG_Property` class is similar, except a state with an ID less than zero represents the `error` state and line 4 is replaced by `assert(state >= 0)`. This instructs JPF to detect a property violation and produce a counter-example trace if the `error` state of the property is reached.

### 4.3. Environment modeling

The process-algebra based models that are supported by our design-level tools can be checked in a straightforward way both in isolation and in combination with other models. In contrast to these, software model checkers such as JPF analyze executable programs, and as such, expect com-

plete programs as input. Therefore, to analyze system components in isolation in an assume-guarantee style, one must provide for each component an appropriate abstract environment that will enable its analysis. In essence, such environments provide stubs for the methods called by the component that are implemented by other components, or drive the execution of a component by calling methods that the component provides to its environment.

In our experiments, we used *universal environments*, that may invoke any provided operation in a component's interface or refuse any required operations in any order. Tools that build such environments for Java programs are presented in [31].

### 4.4. Analysis of implementations

For checking implementations, we instrument the source code to perform assume-guarantee reasoning (as described above) and we use off-the shelf software model checking tools (i.e. Java PathFinder for the analysis of Java programs). Note that, for complex software, even components may be too large to analyze exhaustively; abstraction and slicing techniques (e.g. [10]) could be used to make analysis of software components more tractable. Alternatevely, our approach could sacrifice exhaustiveness for the sake of scalability by using lighter-weight techniques such as stateless model checking [17] or runtime analysis [20].

## 5. Case study

Our case study is the planetary rover controller K9, and in particular its executive subsystem, developed at NASA Ames Research Center. It has been performed in the context of an ongoing collaboration with the developers of the Rover, where verification and development go hand-in-hand to increase the quality of the design and implementation of the system.

In this section we describe how we used our assume-guarantee frameworks to check a key property on the design models of the executive and to automatically generate an appropriate assumption. We show how this assumption was used to perform assume-guarantee model checking of source code with JPF and how this compares to the monolithic (i.e. non-compositional) analysis of the executive's Java implementation.

### 5.1. System description

The executive receives flexible plans from a planner, which it executes according to the plan language semantics. A plan is a hierarchical structure of actions that the Rover must perform. Traditionally, plans are deterministic sequences of actions. However, increased Rover autonomy

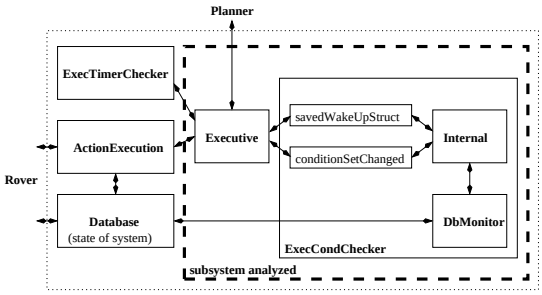


**Figure 7. The Executive of the K9 Mars Rover**

requires added flexibility. The plan language therefore allows for branching based on state or temporal conditions that need to be checked, and also for flexibility with respect to the starting time of an action.

The executive needs to monitor the state of the Rover and of the environment to take appropriate paths in a flexible plan that it executes. It has been implemented as a multi-threaded system (see Figure 7), made up of a main coordinating component named *Executive*, components for monitoring the state conditions *ExecCondChecker*, and temporal conditions *ExecTimerChecker* - each further decomposed into two threads - and finally an *ActionExecution* thread that is responsible for issuing the commands to the Rover. The executive has been implemented as 25K lines of C++ code, 10K of which is the main control code, and the rest defines data structures that are needed for the communication with the actual Rover. The software makes use of the POSIX thread library, and synchronization between threads is performed through mutexes and condition variables.

### 5.2. Design-level analysis

The developers provided their design documents that described the synchronization between components in an ad-hoc flowchart-style notation. These were in essence extended control-flow graphs of the threads, and focused on such things as method calls, (un-)locking mutexes and wait-for and signaling of condition variables. They looked very much like LTSs, which allowed us to translate them in a straightforward and systematic, albeit manual, way into about 700 lines of FSP code. FSP is the input language of the LTSA tool, in which we have implemented our assume-guarantee frameworks described in Section 2. To achieve a close correspondence between the FSP code and the design diagrams, we first built models for mutexes, condition variables, and their associated methods. These models provided an infrastructure on top of which the actual threads of the system were modeled.

Model checking of the design models uncovered a number of synchronization problems such as deadlocks and data races. Moreover, the models were used for quick experimentation with alternative solutions to existing defects. The

study that we present here focuses on the following property that was formulated by the developer.

**Property.** For the variable *savedWakeUpStruct* of the *ExecCondChecker* that is shared with the *Executive* (see Figure 7), the property states that: "if the *Executive* thread reads the value of the variable, then the *ExecCondChecker* should not read this value until the *Executive* clears it first". The property was represented in terms of two states corresponding to the shared variable being cleared or not cleared, and an error state as discussed in the previous section.

**Analysis.** The developer expected the property to be satisfied by the *ExecCondChecker* and the *Executive* irrespective of the behavior of other threads. Our analysis was therefore performed on these threads together with the mutexes they use, since mutexes are the way in which synchronization issues are resolved in the system. We applied assume-guarantee reasoning as supported by our techniques described in Section 2, where assumptions were generated for the *ExecCondChecker* (module $M_1$) and discharged by the *Executive* (module $M_2$).

The weakest assumption consists of 6 states and description an environment where "whenever the *Executive* reads the *savedWakeUpStruct* variable after acquiring mutex *exec*, it should hold on to the mutex until it clears the variable". This assumption could not be discharged on the *Executive*. The counter-example obtained describes the following scenario: if the *Executive* reads *savedWakeUpStruct* and decides that the variable points to an irrelevant condition, it performs a *wait* on a condition variable associated with the *exec* lock. The *wait* causes the *exec* lock to be released automatically. The problem was fixed by adding to the *Executive* a statement that clears *savedWakeUpStruct* before checking whether the condition contained there is relevant.

### 5.3. Implementation analysis

**Set-up.** We analyzed a Java translation of this code, which was used in a case-study described in [5]. The translation was done selectively and it focused on the core functionality of the executive (the rest of the components being stubbed). The translated Java version is approximately 7.2 Kloc and it contains all the components of Figure 7, where each component (except the *Database*) executes as a separate thread.

In our experiment, we concentrated on a subsystem consisting of the *Executive* and *ExecCondChecker* threads (all the other threads were not started), and we analyzed this system for a very simple input plan, that consists of one node and no time conditions (which are not relevant for the analysis of the subsystem). JPF was able explore exhaustively the state-space of this subsystem (any other configuration, i.e. starting more threads or more complex input plans would force JPF to run out of memory). This subsystem is small enough to be manageble by JPF, and thus to provide a baseline for comparison with modular verification, but it still contains enough details about the interactions between the *Executive* and the *ExecCondChecker* threads, which were the focus of the design-level analysis.

To evaluate the merits of assume-guarantee verification using the automatically generated design-level assumption, we broke up the system in two components $I_1$ and $I_2$ representing the ExecCondChecker and the Executive threads respectively, and we checked the two premises of the assume-guarantee rule.

**Environment modeling.** As was mentioned in Section 4, to check components $I_1$ and $I_2$ in isolation, we need to identify the interface between them so we can generate environment models to make them closed.

To check premise 1, we built a universal environment to drive $I_1$ (the *ExecCondChecker*), that invokes any sequence of operations in the class `ExecCondChecker`'s interface (see Figure 8).

This driver loops forever generating events (line 1). It begins by making a non-deterministic choice of whether or not to acquire the lock on the *Executive* object (line 2). If it acquires the lock (line 3), we then use a specialized form the of `AG_Monitor.event` method to trap the lock event (line 4). The universal environment then makes a non-deterministic choice (line 5). Depending on the results of this choice, zero or more events are generated while the lock is held (lines 6-10). These events include method calls that access *savedWakeUpStruct* which is shared between the two threads (i.e. `condChecker.deleteSavedWakeup()` and `condChecker.getSavedWakeup()`) and methods that add and remove conditions to and from a list structure in `ExecCondChecker`. Once the universal environment is done generating events, it generates an event sigaling that the lock is to be released (line 11) and then leaves the synchronized block. If the choice was made to not acquire the lock on line 2, then the universal environment generates a single event (lines 12-16).

To maintain a finite number of elements in the list of conditions, we added an annotation forcing JPF to backtrack if more than one call to `condChecker.addConditionCheck()` is made; this is a reasonable assumption, since we considered a configuration where the input plan has only one node (and only one condition could be added for it).

To check premise 2, we built stubs that implement the methods invoked in component $I_1$ by $I_2$. Some care needed to be taken when doing this. For example, the `getSavedWakeup()` method can either return null or an object. To simulate this, the method stub would non-deterministically choose which to return.

**Analysis.** We used JPF and the property ($P$) and assump-

```
class Executive { ...
    public void run() { ...
1)    while(true) {
2)      if(Verify.randomBool()) {
3)        synchronized(exec) {
4)          AG_Monitor.event("Executive", "lock");
5)          while(Verify.randomBool()) {
6)            switch(Verify.random(4)) {
7)              case 0:  condChecker.deleteSavedWakeup(); break;
8)              case 1:  condChecker.getSavedWakeup(); break;
9)              case 2:  condChecker.addConditionCheck(id,...); break;
10)             case 3:  condChecker.removeConditionCheck(id,...); break;
            } }
11)         AG_Monitor.event("Executive", "unlock"); }
        } else {
12)       switch(Verify.random(4)) {
13)         case 0:  condChecker.deleteSavedWakeup(); break;
14)         case 1:  condChecker.getSavedWakeup(); break;
15)         case 2:  condChecker.addConditionCheck(id,...); break;
16)         case 3:  condChecker.removeConditionCheck(id,...); break;
          } } } } }
```

**Figure 8. Universal driver**

tion ($A$) that were used in the design level analysis to check the property monolithically (i.e., on the whole subsystem) and modularly (i.e. we checked premise 1: $\langle A \rangle\ I_1\ \langle P \rangle$ and premise 2: $\langle true \rangle\ I_2\ \langle A \rangle$). In both cases, we discovered the same error that was discovered at the design level. After we corrected the error, we repeated the checks. While the property was shown to hold on the whole sub-system, we were surprised to find out that premise 1 would not hold, i.e. assumption $A$ was not strong enough to make the property hold. After looking back at the design model, we noticed that the system for which we had generated the assumption also encoded a different assumption, according to which all accesses to *savedWakeupStruct* by the *Executive* thread would be protected by the *exec* lock. This assumption was encoded explicitely at the indications of the developer who gave us the initial models (the assumption was subsequently discharged on $M_2$). Using this new assumption, we checked that the property holds (i.e. we checked that $\langle A \wedge A' \rangle\ I_1\ \langle P \rangle$ holds and we discharged both assumptions on $I_2$).

**Results and discussion.** Our experiments were run on a Intel Xeon 2.2 Ghz machine with 4Gb of memory (although a single process could only access 2Gb of memory). This system is running RedHat Linux version 8.0 with Sun's Java version 1.4.2-01. We used JPF version 2.4 using the -no-verify-print, -no-deadlocks, and -verbose flags.

Table 1 gives the results of the experiment. The System column describes the system being analyzed. The States and Transitions columns report the number of states and transitions explored by JPF. The Memory and Time report the amount of memory needed and the time taken to perform the analysis.

The Whole System rows give the results for checking the property monolithically. The version marked bug corresponds to the original system in which the property does not hold while the other version has had the bug fixed so that the property does hold.

The Premise 1 lines report the results of verifying premise 1. As was mentioned previously, while performing the verification, we discovered that an additional assumption, $A'$ was needed to complete the verification. We looked at two ways of incorporating this assumption into the analysis. The first uses the universal environment shown in Figure 8 and uses an automaton representation of $A'$, as shown in the `AG_Assumption` class in Figure 6. The second uses a modified universal assumption that directly encodes $A'$. This is done by replacing lines 12-16 of the universal assumption with code that makes a choice only between the two events on lines 15 and 16. The bug that caused a violation of the property in the monolithic analysis was in the *Executive*, not the *ExecConditionChecker*, so these analyses were not affected by the presence or absence of the bug.

The Premise 2 lines report the results for checking premise 2, in which the assumptions used in checking Premise 1 need to be discharged. We discharged the assumptions $A$ and $A'$ separately, on the system containing the bug and on the system in which the bug is fixed.

From Table 1, we can see that the compositional approach to verification does reduce the number of states that JPF needs to explore and the amount of memory necessary for the analysis in the version of the Rover that does not contain the bug. The results from checking Premise 1 show

**Table 1. Experimental results**

| System | States | Transitions | Memory (Mb) | Time |
|---|---|---|---|---|
| Whole System | 183,132 | 425,641 | 952.85 | 12m, 24s |
| Whole System (bug) | 255 | 338 | 23.07 | 10s |
| Premise 1, $A'$ as automaton | 60,830 | 134,177 | 315.98 | 6m, 55s |
| Premise 1, $A'$ encoded | 53,215 | 117,756 | 255.96 | 4m, 49s |
| Premise 2, Assumption $A$ | 13,884 | 20,601 | 118.97 | 1m, 16s |
| Premise 2, Assumption $A$ (bug) | 145 | 144 | 44.49 | 20s |
| Premise 2, Assumption $A'$ | 13,884 | 20,601 | 109.58 | 1m, 7s |
| Premise 2, Assumption $A'$ (bug) | 13,884 | 20,601 | 121.37 | 49s |

that the encoding of the assumption can affect the performance of the model checker. We plan to investigate this in the future.

This case study demonstrates that the use of design level assumptions has merits in improving the performance of source code model checking. Our experimental work is of course preliminary, and we are planning to carry out larger case studies to validate our approach.

## 6. Related work

It is well known that software defects are less costly the earlier they are removed in the development process. Towards this end, a number of researchers have worked on applying model checking to artifacts that appear throughout the software life-cycle, such as requirements [3, 21], architectures and designs [1, 27, 29] and source code [4, 6, 10, 13, 32]. Our work integrates the analysis performed at different levels, using assume-guarantee reasoning.

Assume-guarantee reasoning is based on the observation that large systems are being build from components and that this composition can be leveraged to improve the performance of analysis techniques. Formal techniques for support of component-based design are gaining prominence, see for example [11]. To reason formally about components in isolation, some form of assumption (either implicit or explicit) about the interaction with, or interference from, the environment has to be made. Even though we have sound and complete reasoning systems for assume-guarantee reasoning, see for example [8, 19, 25, 30], it is always a mental challenge to obtain the most appropriate assumption.

It is even more of a challenge to find automated techniques to support this style of reasoning. The thread modular reasoning underlying the Calvin tool [14] is one start in this direction. The Mocha toolkit [2] provides support for modular verification of components with requirement specifications based on the Alternating-time Temporal logic.

More recently, Henzinger et al. [22] have presented a framework for thread-modular abstraction refinement, in which assumptions and guarantees are both refined in an iterative fashion. The framework applies to programs that communicate through shared variables, and, unlike our approach where assumptions are controllers of the component that is being analyzed, the assumptions in [22] are abstractions of the environment components. The work of Flanagan and Qadeer also focuses on a shared-memory communication model [15], but does not address notions of abstractions as is done in [22] . Jeffords and Heitmeyer use an invariant generation tool to generate invariants for components that can be used to complete an assume-guarantee proof [24]. While their proof rules are sound and complete, their invariant generation algorithm is not guaranteed to produce invariants that will complete an assume-guarantee proof even if such invariants exist.

## 7. Conclusions

We presented an approach for integrating assume-guarantee verification at different phases of system development, to address the scalability issues associated with the verification of complex software systems. Our approach uses the results of modular analysis at the design level to improve the performance of verification at the code level. We gave a program instrumentation technique for supporting assume-guarantee reasoning of Java programs using the Java PathFinder model checker; our approach easily extends to other programmming languages and model checkers. We also presented a significant case study demonstrating the applicability of our approach to a realistic NASA software system. To evaluate how useful our approach is in practice, we are planning its extensive application to other real systems. However, our early experiments provide strong evidence in favor of this line of research.

In the future, we plan to look at ways to better automate the process of code annotation and environment generation. Additionally, we plan to investigate lighter-weight techniques such as stateless model checking or run-time verification in the context of our methodology. Finally, we plan to evaluate the use of other design-level artifacts to improve the performance of verification at source-code level.

## References


[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. on Software Eng. and Methodology*, 6(3):213–249, July 1997.

[2] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proc. of the Tenth Int. Conf. on Computer-Aided Verification*, pages 521–525, June 28–July 2, 1998.

[3] J. M. Atlee and J. D. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. on Software Eng.*, 19(1):24–40, Jan. 1993.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of the Eighth SPIN Workshop*, pages 101–122, May 2001.

[5] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Păsăreanu, A. Venet, and W. Visser. Experimental evaluation of V&V tools on Martian rover software. In *SEI Software Model Checking Workshop*, Mar. 2003.

[6] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. of the* $25^{th}$ *Int. Conf. on Software Eng.*, pages 385–395, May 2003.

[7] Y.-P. Cheng, M. Young, C.-L. Huang, and C.-Y. Pan. Towards scalable compositional analysis by refactoring design models. In *Proc. of the Ninth European Software Eng. Conf. held jointly with the Eleventh ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 247–256, Sept. 2003.

[8] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Proc. of the Fourth Symp. on Logic in Comp. Sci.*, pages 353–362, June 1989.

[9] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. of the Ninth Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346, Apr. 2003.

[10] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the* $22^{nd}$ *Int. Conf. on Software Eng.*, June 2000.

[11] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. of the Eighth European Software Eng. Conf. held jointly with the Ninth ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 109–120, Sept. 2001.

[12] D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software architecture themes in JPL's mission data system. In *2000 IEEE Aerospace Conference*, Mar. 2000.

[13] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proc. of the Second ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 62–75, Dec. 1994.

[14] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. of the Eleventh European Symp. on Programming*, pages 262–277, Apr. 2002.

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

[16] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the Seventeenth IEEE Int. Conf. on Automated Software Eng.*, Sept. 2002.

[17] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of the* $24^{th}$ *ACM Symp. on Principles of Programming Languages*, pages 174–186, Jan. 1997.

[18] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proc. of the 2002 Int. Symp. on Software Testing and Analysis*, pages 12–21, July 2002.

[19] O. Grumberg and D. E. Long. Model checking and modular verification. In *Proc. of the Second Int. Conf. on Concurrency Theory*, pages 250–265, Aug. 1991.

[20] K. Havelund and G. Roşu. Java pathexplorer - a runtime verification tool. In *Proc. of the Sixth Int. Symp. on Artificial Intelligence, Robotics, and Automation in Space*, 2001.

[21] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. on Software Eng. and Methodology*, 5(3):231–261, July 1996.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. of the Fifteenth Int. Conf. on Computer-Aided Verification*, pages 262–274, July 2003.

[23] iContract home page. http://www.reliable-systems.com/tools/iContract/iContract.htm.

[24] R. D. Jeffords and C. L. Heitmeyer. A strategy for efficiently verifying requirements. In *Proc. of the Ninth European Software Eng. Conf. held jointly with the Eleventh ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 28–37, Sept. 2003.

[25] C. B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.

[26] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.

[27] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Proc. of the First Working IFIP Conf. on Software Architectures*, Feb. 1999.

[28] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[29] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Applying static analysis to software architectures. In *Proc. of the Sixth European Software Eng. Conf. held jointly with the Fifth ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 77–93, Sept. 1997.

[30] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, New York, 1984. Springer-Verlag.

[31] O. Tkachuk, M. B. Dwyer, and C. Păsăreanu. Automated environment generation for software model checking. In *Proc. of the Thirteenth IEEE Int. Conf. on Automated Software Eng.*, 2003.

[32] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the Fifteenth IEEE Int. Conf. on Automated Software Eng.*, pages 3–12, Sept. 2000.