

# Assume-Guarantee Testing

Colin Blundell

Dept. of Comp. and Inf. Science  
University of Pennsylvania  
Philadelphia, Pennsylvania USA

blundell@cis.upenn.edu

Dimitra Giannakopoulou

RIACS/NASA Ames  
NASA Ames Research Center  
Moffett Field, CA 94035-1000, USA

dimitra@email.arc.nasa.gov

Corina S. Păsăreanu

QSS/NASA Ames  
NASA Ames Research Center  
Moffett Field, CA 94035-1000, USA

pcorina@email.arc.nasa.gov

## ABSTRACT

Verification techniques for component-based systems should ideally be able to predict as many properties as possible of the assembled system through local properties of the system components. This paper introduces such a technique in the context of testing. Assume-guarantee testing is based on the (automated) decomposition of key system-level requirements into local component requirements at design time. The latter are checked during testing of individual components, and can thus uncover violations of system requirements before system assembly. Moreover, when a system is assembled, we provide assume-guarantee techniques that can efficiently predict, based on correct system runs, violations by alternative system runs. We discuss the application of our approach to testing a multi-threaded NASA application, where we treat threads as components.

## Keywords

verification, testing, assume-guarantee reasoning, predictive analysis

## 1. INTRODUCTION

As software systems continue to grow in size and complexity, it is becoming common for developers to assemble them from new or reused components potentially developed by different parties. For these systems, it is important to have verification techniques that are modular as well, since verification is often the dominant cost in money and time. Developers could use such techniques to avoid expensive whole-system verification, instead performing verification primarily on individual components. Unfortunately, the task of extracting useful results from verification of components in isolation is often difficult: first, developing environments that will appropriately exercise individual components is challenging and time-consuming, and second, inferring system properties from results of local verification is typically non-trivial. The growing popularity of component-based systems makes it important for verification researchers to investigate these

challenges.

Assume-guarantee reasoning is a technique that has long held promise for modular verification. This technique is a “divide-and-conquer” approach that infers global system properties by checking individual components in isolation [4, 12, 14, 16]. In its simplest form, it checks whether a component  $M$  guarantees a property  $P$  when it is part of a system that satisfies an assumption  $A$ , and checks that the remaining components in the system ( $M$ 's environment) satisfy  $A$ ; it can also be extended to deal with the verification of multiple components and the use of assumptions for the verification of each of those. In previous work, we developed techniques that *automatically* generate assumptions for performing assume-guarantee model checking at the design level [2, 5, 9].

Of course, verifying the design is not enough; we must also verify that the implementation preserves the design's correctness. For this purpose, we have also previously developed a methodology that uses the assumptions created at the design level to model check source code in an assume-guarantee style, as presented in [10]. Similarly to the design-level verification, source code is thus also checked one component at a time. Hence, this technique has potential to meet the challenges of component-based verification.

Unfortunately, despite the increased scalability that one can achieve by using assume-guarantee techniques in model checking, it remains a difficult task in the hands of experts to make the technique scale to the size of industrial systems. Furthermore, model checkers do not exist for many industrial programming languages. In this work, we explore the benefits of assume-guarantee reasoning for testing, which is still the predominant industrial verification technique. Assume-guarantee testing uses design-level assumptions: — as testing environments for individual system components, and — to perform predictive analysis of the assembled system. We believe that this approach enhances the testing process in several ways:

1. Assumptions can provide appropriate testing environments. They restrict the context of the components, thus reducing the number of false positives obtained by verification (false positives are errors that will never exhibit themselves in the context of the particular system in which the component will be introduced). Violations of system-level properties can thus be detected during component testing.

This is desirable, as it is well established that errors discovered earlier in the development phase are easier and cheaper to fix. We realize that components must sometimes be thoroughly tested irrespective of their context, as is the case for component libraries. However, it is equally important to be able to customize the testing process when particular assemblies are considered.

2. Assume-guarantee testing has the potential to obtain system coverage through local testing. We apply assume-guarantee reasoning to component testing-traces in order to obtain similar guarantees on trace compositions as we previously obtained on component compositions by using assume-guarantee reasoning in the context of model checking. Hence, we can infer that a (potentially large) set of system traces satisfies a property by checking traces of components in isolation against assume-guarantee pairs.

3. Assume-guarantee testing also has the potential to more efficiently detect bugs and provide coverage during whole-system testing, which will remain an important part of system verification for the foreseeable future. Here, our technique is an efficient means of predictive testing, in which the existence of bad traces is detected from a good trace. Predictive testing exploits the insight that within a trace, events known to be independent can be reordered to obtain different legal traces. Typically, predictive testing techniques discover these alternative interleavings by composing independent events in different orders [18]; we use assume-guarantee reasoning to obtain results about the alternative interleavings without explicitly exploring them.

We experimented with our assume-guarantee testing framework in the context of the Eagle runtime analysis tool [3], and applied our approach to a NASA software system also used in the demonstration of our design-level assume-guarantee reasoning techniques. In the analysis of a specific property ( $P$ ) during these experiments, we found a discrepancy between one of the components and the design that it implements. This discrepancy does not cause the system to violate  $P$  and would therefore not have been detected during monolithic model checking of  $P$ .

The remainder of the paper is organized as follows. We first provide some background in Section 2, followed by a discussion of our assume-guarantee testing approach and its advantages in Section 3. Section 4 describes the experience and results obtained by the application of our techniques to a NASA system. Finally, Section 5 presents related work and Section 6 concludes the paper.

## 2. BACKGROUND

**LTSs.** At design level, we use Labeled Transition Systems (LTSs) to model the behavior of communicating components. Let  $\mathcal{Act}$  be the universal set of observable actions and let  $\tau$  denote a local action *unobservable* to a component's environment. An LTS  $M$  is a quadruple  $\langle Q, \alpha M, \delta, q_0 \rangle$  where:

- $Q$  is a non-empty finite set of states
- $\alpha M \subseteq \mathcal{Act}$  is a finite set of observable actions called the *alphabet* of  $M$

- $\delta \subseteq Q \times \alpha M \cup \{\tau\} \times Q$  is a transition relation
- $q_0 \in Q$  is the initial state

Let  $M = \langle Q, \alpha M, \delta, q_0 \rangle$  and  $M' = \langle Q', \alpha M', \delta', q_0' \rangle$ . We say that  $M$  *transits* into  $M'$  with action  $a$ , denoted  $M \xrightarrow{a} M'$ , if and only if  $(q_0, a, q_0') \in \delta$  and  $\alpha M = \alpha M'$  and  $\delta = \delta'$ .

**Traces.** A *trace*  $t$  of an LTS  $M$  is a sequence of observable actions that  $M$  can perform starting at its initial state. For  $\Sigma \subseteq \mathcal{Act}$ , we use  $t|\Sigma$  to denote the trace obtained by removing from  $t$  all occurrences of actions  $a \notin \Sigma$ . The set of all traces of  $M$  is called the *language* of  $M$ , denoted  $\mathcal{L}(M)$ .

Let  $t = \langle a_1, a_2, \dots, a_n \rangle$  be a finite trace of some LTS  $M$ . We use  $[t]$  to denote the LTS  $M_t = \langle Q, \alpha M, \delta, q_0 \rangle$  with  $Q = \{q_0, q_1, \dots, q_n\}$ , and  $\delta = \{(q_{i-1}, a_i, q_i)\}$ , where  $1 \leq i \leq n$ .

**Parallel Composition.** The parallel composition operator  $\parallel$  is commutative and associative. It combines the behavior of two components by synchronizing the actions common to their alphabets and interleaving the remaining actions. Formally, let  $M_1 = \langle Q_1, \alpha M_1, \delta_1, q_0_1 \rangle$  and  $M_2 = \langle Q_2, \alpha M_2, \delta_2, q_0_2 \rangle$  be two LTSs. Then  $M_1 \parallel M_2$  is an LTS  $M = \langle Q, \alpha M, \delta, q_0 \rangle$ , where  $Q = Q_1 \times Q_2$ ,  $q_0 = (q_0_1, q_0_2)$ ,  $\alpha M = \alpha M_1 \cup \alpha M_2$ , and  $\delta$  is defined as follows, where  $a$  is either an observable action or  $\tau$  (note that the symmetric rules are implied by the fact that the operator is commutative):

$$\frac{M_1 \xrightarrow{a} M'_1, a \notin \alpha M_2}{M_1 \parallel M_2 \xrightarrow{a} M'_1 \parallel M_2}$$

$$\frac{M_1 \xrightarrow{a} M'_1, M_2 \xrightarrow{a} M'_2, a \neq \tau}{M_1 \parallel M_2 \xrightarrow{a} M'_1 \parallel M'_2}$$

**Properties and Satisfiability.** A property is specified as an LTS  $P$ , whose language  $\mathcal{L}(P)$  defines the set of acceptable behaviors over  $\alpha P$ . An LTS  $M$  satisfies  $P$ , denoted as  $M \models P$ , if and only if  $\forall t \in \mathcal{L}(M). t|\alpha P \in \mathcal{L}(P)$ .

**Assume-guarantee Triples.** 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 [16]. The formula is true if whenever  $M$  is part of a system satisfying  $A$ , then the system guarantees  $P$ . At design level in our framework, all of  $A, M, P$  are expressed as LTSs.

**Assume-guarantee Reasoning.** Consider for simplicity a system that is made up of components  $M_1$  and  $M_2$ . The aim of assume-guarantee reasoning is to establish  $M_1 \parallel M_2 \models P$  without composing  $M_1$  with  $M_2$ . The simplest assume-guarantee proof rule that can be used for this purpose consists of showing that the following two premises hold:  $\langle A \rangle M_1 \langle P \rangle$  and  $\langle true \rangle M_2 \langle A \rangle$ . From these, the rule infers that  $\langle true \rangle M_1 \parallel M_2 \langle P \rangle$  also holds. 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  $P$ .

In previous work we developed frameworks that compute assumptions automatically for finite-state models and safety properties that are expressed in terms of LTSs. More specifically, in [9] 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. In [5] and [2], a learning algorithm is used to compute assumptions in an *incremental* fashion in the context of simple and symmetric assume-guarantee rules, respectively.

### 3. ASSUME-GUARANTEE TESTING

This section describes our methodology for using the artifacts of the design-level analysis, i.e. models, properties and generated assumptions, for testing the implementation of a software system. We assume a top-down software development process, where one creates and debugs design models, which are then used to guide the development of source code, possibly by (semi-) automatic code synthesis.

Our approach is illustrated by Figure 1. Consider a system that consists of two (finite-state) design models  $M_1$  and  $M_2$ , and a global safety property  $P$ . Assume that the property holds at the design level (if the property does not hold, we use the feedback provided by the verification framework to correct the models until the property is achieved). The assume-guarantee framework that is used to check that the property holds will also generate 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), as described in Section 2.

Once the property is established at the design level, we need to check if the property holds at the implementation level, i.e. if  $C_1 \parallel C_2 \models P$ . A close correspondence between the design-level components and the actual software implementations<sup>1</sup>, e.g. components  $C_1$  and  $C_2$  implement  $M_1$  and  $M_2$ , respectively, in Figure 1. We propose *assume-guarantee testing* as a way of checking  $C_1 \parallel C_2 \models P$ . This consists of producing test traces by each of the two components, and checking these traces against the respective assume-guarantee premises applied at the design level. If each of the checks succeeds, then we are guaranteed that the composition of the traces satisfies the property  $P$ .

We illustrate assume-guarantee testing through a simple example. Consider a communication channel that has two components, designs  $M_1$  and  $M_2$  and corresponding code  $C_1$  and  $C_2$  (see Figure 2). Property  $P$  describes all legal executions of the channel in terms of events *in, out*; it essentially states that trace  $\langle out, in \rangle$  is not allowed, since this trace is the only  $P$  trace that is missing from  $P$ . Figure 2 also shows the assumption  $A$  that is generated during the design-level analysis of  $M_1 \parallel M_2$  (see Section 2). Note that, although  $M_1 \parallel M_2 \models P$ ,  $C_1 \parallel C_2$  violates the property. Figure 3 (left) shows two traces  $t_1$  and  $t_2$ , ob-

<sup>1</sup>The software architecture of a system may not always provide the best decomposition for verification; however, for the NASA systems that we have studied, the architectural decomposition lends itself well to our automated assume-guarantee framework.

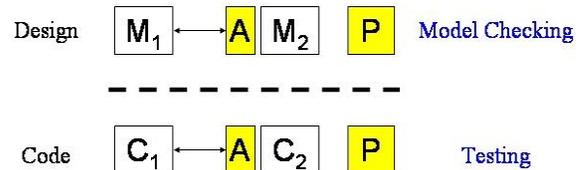


Figure 1: Design and code level analysis

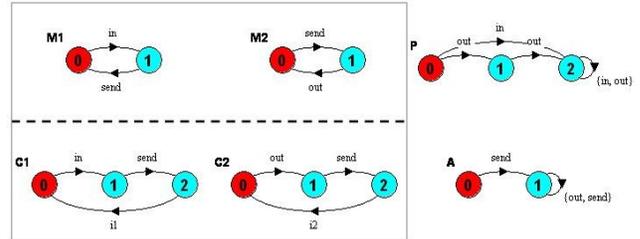


Figure 2: Assume-guarantee testing

tained from components  $C_1$  and  $C_2$ , respectively. Checking  $\langle true \rangle t_2 \langle A \rangle$  during assume-guarantee testing will detect the fact that  $t_2$  violates the assumption  $A$  and will thus uncover the problem with the implementation. Assume now that assume-guarantee testing is not used, but the assembled system is tested instead (we call the latter monolithic testing). The system might first produce the first two traces illustrated in Figure 3 (right). These traces satisfy the property, which could lead developers to mistakenly believe that the system is correct. This belief could even be supported by the achievement of some testing coverage criterion, as discussed later in this section.

In summary, the contribution of assume-guarantee testing is that we can obtain results on all interleavings of two individual component traces simply by checking each against the appropriate assume-guarantee premise. In the context of our example, checking  $t_1$  and  $t_2$  corresponds to checking all four traces illustrated in Figure 3 (right).

While our example illustrates the benefits of assume-guarantee reasoning for unit testing, similar benefits apply to whole-system testing. When the system is assembled, we use assume-guarantee testing to conduct analysis that can efficiently predict, based on correct system runs, violations by alternative system runs. Both flavors of assume-guarantee testing are discussed in more detail below.

#### 3.1 Assume-Guarantee Component Testing

The first step in assume-guarantee testing involves the implementation of 1)  $U_A$  for  $C_1$ , where  $U_A$  encodes  $C_1$ 's universal environment restricted by assumption  $A$ , and 2) the universal environment  $U$  for  $C_2$ . The universal environment for a component may exercise any service that the component provides in any order, and may provide or refuse any service that the component requires. Subsequently, traces are obtained by executing each component in its designated environment. More specifically, a set of traces  $T_1$  is obtained by executing  $C_1$  in  $U_A$  several times and for a variety of test inputs, and similarly  $T_2$  is obtained by executing  $C_2$  in  $U$ . Assume-guarantee testing is then performed for these

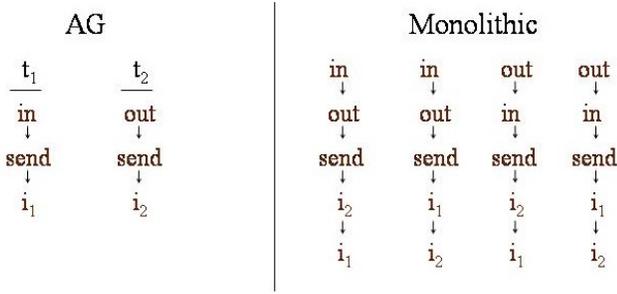


Figure 3: Discovering bugs with fewer tests

traces. Each trace  $t_1 \in T_1$  is checked against  $P$ , and each trace  $t_2 \in T_2$  is checked against  $A$ . If either of these checks fail (as in Figure 3), this is an indication that there is an incompatibility between the models and the implementations, and the implementation or the model (or both) should be corrected. If all these tests succeed, then we know by the assume-guarantee rule that  $[t_1] \parallel [t_2] \models P$ , for all  $t_1 \in T_1$  and  $t_2 \in T_2$ .

An advantage of this approach is that system correctness can be established through local tests of components. Such tests can be performed as soon as each component becomes “code complete”, and *before* other components have been implemented, or assembled together to form an executable system. A secondary advantage of this approach is that it ameliorates the problem of choosing appropriate testing environments for components in isolation. This is a difficult problem in general, as finding an environment that is both general enough to fully exercise the component under testing and specific enough to avoid many false positives is usually a time-consuming iterative process. Here, this problem is reduced to that of correctly implementing  $U_A$  and  $U$ . Note that 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. Therefore, checking the assumptions can be done more efficiently. Finally, note that, when checking components in isolation, one has more control over the component interface (since it is exercised directly rather than through some other component). As a result, it is both easier to reproduce problematic behavior, and to exercise more traces for constructing sets  $T_1$  and  $T_2$ .

**Coverage.** Unlike model checking, testing is not an exhaustive verification technique. As a result, it is possible for defects to escape despite testing. For this reason, the notion of *coverage* has traditionally been associated with the technique. Coverage criteria dictate how much testing is “enough” testing. A typical coverage criterion that works on the structure of the code is “node” coverage, which requires that the tests performed cover all nodes in the control flow graph of the implementation of a system. Assume that in our example our coverage criterion is node coverage for  $C_1$  and  $C_2$ . Then  $t_1$  and  $t_2$  in Figure 3 (left) together achieve 100% coverage. Similarly, the first trace of the assembled system in Figure 3 (right) achieves 100% node coverage. It is therefore obvious that assume-guarantee testing has the po-

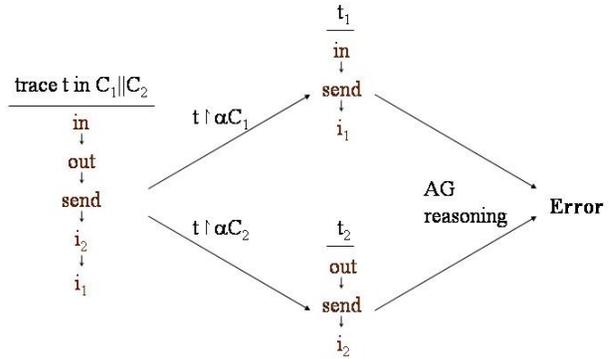


Figure 4: Predictive analysis

tential of checking more behaviors of the system even when it achieves the same amount of coverage. This example also reflects the fact that traditional coverage criteria are often not appropriate for concurrent or component-based systems, which is an area of active research. One could also measure coverage by the number of behaviors or paths through the system that are exercised. In this case, assume-guarantee testing has the potential of achieving the same coverage as monolithic testing with fewer tests.

**Discussion.** As stated above, our hope is that by checking individual traces of components, we essentially cover multiple traces of the assembled system. Unfortunately, this is not always true, due to the problem of *incompatible traces*, which are traces that do not execute the same shared events in the same order. These traces are from different execution paths, and thus cannot be composed to get useful knowledge about the system (they give the empty trace when composed.) For example, suppose that the first event in  $t_1$  is a function call on the procedure `foo` in  $C_1$ , while the first event in  $t_2$  is a function call on the procedure `bar` in  $C_2$ ; these traces executed on different paths and are incompatible. Thus, we face the question of producing compatible traces during component testing. However, since multiple traces are produced when constructing sets  $T_1$  and  $T_2$ , we believe that enough compatible traces will typically exist in the cross-product of the two sets. One potential approach would be to use the component models as a coverage metric when generating traces in  $T_1$  and  $T_2$ , and require that each set of traces cover certain sequences of shared events in the models.

### 3.2 Predictive Analysis for Component Assemblies

We may also use assume-guarantee testing to achieve predictive analysis of component assemblies. Assume-guarantee testing for predictive analysis has the following steps:

- Obtain a system trace  $t$  (by running  $C_1 \parallel C_2$ ).
- Project the trace on the alphabets of each component; obtain  $t_1 = t|_{\alpha C_1}$  and  $t_2 = t|_{\alpha C_2}$ .
- Use the design level assumption to study the composition of the projections; i.e. check that  $\langle A \rangle [t_1] \langle P \rangle$  and  $\langle true \rangle [t_2] \langle A \rangle$  hold, using model checking.

The approach is illustrated in Figure 4: on the right, we show a trace  $t$  of  $C_1||C_2$  that does not violate the property. On the left, we show the projections  $t_1$  and  $t_2$ . Note that  $\langle true \rangle [t_2] \langle A \rangle$  does not hold, hence from a single “good” trace we have been able to predict that  $C_1||C_2$  violates the property. Notice that using the design level assumption to analyze the projections is more efficient than composing the projections and checking that the composition satisfies the property as is performed by other predictive analysis techniques — this is especially so when the design level assumptions are small.

An alternative approach is to generate the assumption directly from the projected trace  $t_1$ , and then test that  $t_2$  satisfies this assumption. This approach is a way to do assume-guarantee predictive testing in a system where there are no design-level models. However, we have to experiment to see if it is practical to generate a new assumption for each trace.

**Discussion.** We would like to use assume-guarantee predictive testing as a means of efficiently generating system coverage. This technique does not suffer from incompatible traces, as the two projected traces occur in the same system trace and are thus guaranteed to be compatible. Of course, when generating traces of the system, the benefits of predictive analysis should ideally be taken into account. For example, suppose that we obtain a trace  $t$ , projected onto  $t_1$  and  $t_2$ . By assume-guarantee testing we know that  $[t_1]||[t_2] \models P$ . In obtaining further traces, we would like to avoid traces in  $[t_1]||[t_2]$  since these are covered by our assume-guarantee checking of  $t_1$  and  $t_2$ . Again, we can consider using the design-level model as a coverage metric; two traces that have different sequences of shared events through the model will project onto different traces. We could also use test input generation techniques for this purpose. We need to investigate this direction further.

## 4. EXPERIENCE

Our case study is the planetary rover controller K9, and in particular its executive subsystem, developed at NASA Ames Research Center. The study has been performed in the context of an ongoing collaboration with the developers of the rover, where verification was performed *during* development to increase the quality of the design and implementation of the system. Below we describe the rover executive, our design-level analysis, how we used the assumptions generated by this analysis to conduct assume-guarantee testing, and results of this testing.

### 4.1 K9 Rover Executive Subsystem

The executive sub-system commands the rover through the use of high-level *plans*, which it interprets and executes in the context of the execution environment. The executive monitors execution of plan actions and performs appropriate responses and cleanup when execution fails.

The executive has been implemented as a multi-threaded system (see Figure 5), made up of a main coordinating component named *Executive*, components for monitoring temporal conditions *ExecTimerChecker* and state conditions *ExecCondChecker* - which is further decomposed into two threads - and an *ActionExecution* thread that is responsible for issuing the commands (actions) to the rover. The commu-

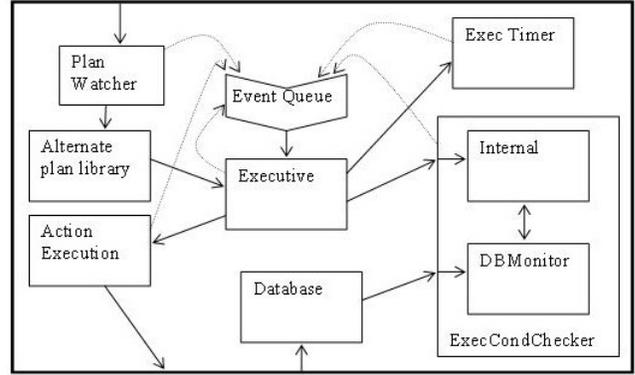


Figure 5: The Executive of the K9 Mars Rover

nication between different components (threads) is made through an *EventQueue*. The implementation has 35K lines of C++ code and it uses the POSIX thread library.

### 4.2 Design-level Analysis

In previous work [9], we developed detailed design models for the executive subsystem. We then checked these models in an assume-guarantee manner for several properties specified by the developers. Model checking of the design models uncovered a number of synchronization problems such as deadlocks and data races, which we then fixed in collaboration with the developers. After finishing this process, for each property we had an assumption on one of the components stating what behavior was needed of it for the property to hold of the entire system.

### 4.3 Assume-guarantee Testing Framework

We have developed a framework that uses the assumptions and properties built during the design level analysis for the assume-guarantee testing of the executive implementation. In order to apply assume-guarantee testing, we broke up the implementation into two components, with the *Executive* thread, the *EventQueue* and the *ActionExecution* thread on one side ( $M_1$ ), and the *ExecCondChecker* thread and the other threads on the other side ( $M_2$ ), as shown in Figure 5.

To test the components in isolation, we generated *environments* that encode the design-level assumptions (as described in Section 3). These environments provide stubs for the methods called by the component that are implemented by other components and drive the execution of a component by calling methods that the component provides to its environment. To make function calls on the component, we provided dummy values of irrelevant arguments (while ensuring that these dummy values did not cause any loss of relevant information).

Essentially, each environment has been implemented as a thread running a state machine (i.e. the respective design level assumption) that executes in an infinite loop, which makes random choices to perform an “active” event that is enabled in the current state, together with a set of functions that comprise the “passive” events of the assumption (these functions also cause the state machine to make the appropriate transition when called by the component under

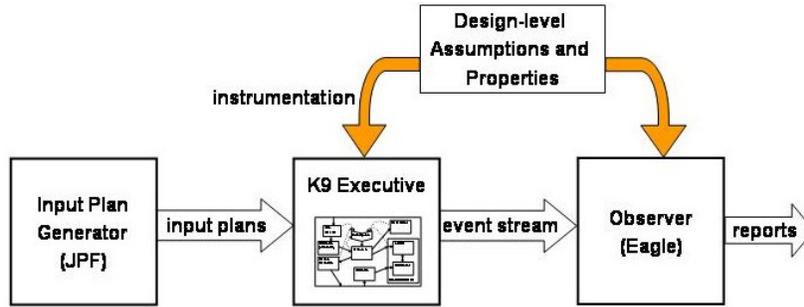


Figure 6: Testing Environment

test).

The framework uses the Eagle run-time monitoring framework [3], to check that the components conform with the assume-guarantee pairs. Eagle is an advanced testing framework that provides means for constructing test oracles that examine the internal computational status of the analyzed system. For run-time monitoring, a program is *instrumented* to emit events which are then checked by Eagle to see whether the current trace conforms to formalized requirements, stated as temporal logic assertions or finite-state automata.

For our experiments, we instrumented (by hand) the code of the executive components, to emit events that appear in the design level assumptions and properties. We translated (automatically) these assumptions and properties into Eagle monitors.

Note that in order to run the executive system (or its components), we need to provide an input plan and an environment simulating the actual rover hardware. For our assume-guarantee testing experiments, the hardware environment was stubbed out. For plan input generation, we built upon our previous work, which combines model checking and symbolic execution for specification based test input generation [20]. To generate test input plans, we encoded the plan language grammar as a nondeterministic specification. Running model checking on this model generates hundreds of input plans in a few seconds.

We have integrated all the above techniques to perform assume-guarantee testing on the executive (see Figure 6). We first instrument the code and generate Eagle monitors encoding design-level assumptions and properties. The framework generates a set of test input plans, a script runs the executive on each plan and it calls Eagle to monitor the generated run-time traces. The user can choose to perform a whole-program (monolithic) analysis or to perform assume-guarantee reasoning.

#### 4.4 Results

We ran several experiments (according to different input plans). For one property, we found a discrepancy between the implementation and the models. The property ( $P$ ) states that the *ExecCondChecker* should not push events onto the *EventQueue* unless the *Executive* has sent the *ExecCondChecker* conditions to check. The design-level assumption ( $A$ ) on the *ExecCondChecker* states that the property will

hold as long as the *ExecCondChecker* sets a flag variable to 1 before pushing events, since these assignments only happen in response to the *Executive* sending conditions.

To check this property, we generated an environment that drives component  $C_1$  (which contains the *Executive*) according to assumption  $A$ . We instrumented  $C_1$  to emit relevant events and we ran Eagle to check if the generated traces conform to property  $P$ .

We also generated a universal environment for component  $C_2$  (which contains the *ExecCondChecker*); we instrumented  $C_2$  to emit events and we used Eagle to check if the generated traces conform to  $A$ . In fact, component  $C_2$  did not conform with the assumption. The obtained counterexample traces exhibit a scenario where the *ExecCondChecker* pushes events onto the *EventQueue* without first setting the flag variable to 1. This turned out to be due to the fact that an input plan can contain null conditions. Instead of putting these in the condition list for monitoring, the *ExecCondChecker* immediately pushes an event to the queue. This behavior exposed an inconsistency between the models and the implementation, which we corrected. Monolithic model-checking of the property  $P$  would not have uncovered this inconsistency.

## 5. RELATED WORK

Assume-guarantee reasoning leverages the observation that large systems are built from components, which can be analyzed in isolation, to improve performance. Formal techniques and tools for support of component-based design and verification are gaining prominence, see for example [1, 6, 8]. All these approaches use some form of environment assumptions (either implicit or explicit), to reason about components in isolation.

In previous work [10] we developed a technique for using design-level assumptions for compositional analysis of source code. In [10] we used model checking (Java PathFinder [19]), while the focus here is on testing. Model checking (the VeriSoft state-less model checker [11]) is also used in [7] for performing assume-guarantee verification for C/C++ components. However, the burden of generating assumptions is on the user.

Our work is related to specification-based testing. There is a lot of work in this area. For example, in [13, 17], formal specifications are used for the generation of test inputs

and oracles. Test inputs are generated from constraints (or assumptions) on the environment of a software component, while test oracles are generated from the guarantees of the component under test. None of the above described approaches address predictive analysis.

Predictive runtime analysis of multithreaded programs has been explored by Sen et al. [18]. Their work uses a partial ordering on events to extract alternative interleavings that are consistent with the observed interleaving; states from these interleavings form a lattice that is similar to our composition of projected traces. However, to verify that no bad state exists in this lattice, they construct the lattice level by level, while we propose using assume-guarantee reasoning to give similar guarantees without having to explore the composition of the projected traces.

Another related approach [15] uses assume-guarantee reasoning to combine runtime monitors. Unlike our work, which aims at improving testing, the goal in [15] is to combine monitoring for diverse features, such as memory management, security and temporal properties, in a reliable way.

## 6. CONCLUSIONS AND FUTURE WORK

We have presented assume-guarantee testing, an approach that improves traditional testing of component-based systems by predicting violations of system-level requirements both during testing of individual components and during system-level testing.

We have experimented with our approach in the verification of a non-trivial NASA system and report promising results. Although we have strong reasons to expect that this technique can significantly improve the state-of-the-art in testing, it is a difficult task to quantify its benefits. One reason is the lack of appropriate coverage criteria for concurrent and component-based systems. Our plans for future work include coming up with “component-based” testing coverage criteria, i.e. criteria which, given the decomposition of global system properties into component properties, determine when individual components have been tested enough to guarantee correctness of their assembly. One potential approach is to use the models as a coverage metric, which is something we wish to investigate in the future.

## REFERENCES

- [1] 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 Comp.-Aided Verification (CAV)*, pages 521–525, June 28–July 2, 1998.
- [2] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Int. Workshop on Specification and Verification of Component-Based Sys.*, Sept. 2003.
- [3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Int. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 44–57, Jan. 2004.
- [4] 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.
- [5] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *9th International Conference for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *LNCS*, Warsaw, Poland, 2003. Springer.
- [6] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. of the Eighth European Soft. Eng. Conf. held jointly with the Ninth ACM SIGSOFT Symp. on the Found. of Soft. Eng.*, pages 109–120, Sept. 2001.
- [7] J. Dingel. Computer Assisted Assume Guarantee Reasoning with VeriSoft. In *International Conference on Software Engineering*, 2003.
- [8] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Proc. of the Eleventh European Symp. on Prog.*, pages 262–277, Apr. 2002.
- [9] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the Seventeenth IEEE Int. Conf. on Auto. Soft. Eng.*, Sept. 2002.
- [10] D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Int. Conf. on Soft. Eng.*, pages 211–220, May 2004.
- [11] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of the 24<sup>th</sup> ACM Symp. on Principles of Prog. Lang.*, pages 174–186, Jan. 1997.
- [12] 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.
- [13] L. J. Jagadeesan, A. A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based testing of reactive software: Tools and experiments (experience report). In *International Conference on Software Engineering*, pages 525–535, 1997.
- [14] 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.
- [15] J. Levy, H. Saidi, and T. E. Uribe. Combining monitors for run-time system verification. *Electronic Notes in Theoretical Computer Science*, 70(4), December 2002.
- [16] 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.

- [17] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *IEEE Real Time Systems Symposium (RTSS)*, 1998.
- [18] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems*, pages 211–226, 2005.
- [19] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the Fifteenth IEEE Int. Conf. on Auto. Soft. Eng.*, pages 3–12, Sept. 2000.
- [20] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *Int. Symp. on Soft. Testing and Analysis*, pages 97–107, July 2004.