Assume-Guarantee Testing

Colin Blundell
Dept. of Comp. and Inf. Science
University of Pennsylvania
Philadelphia, PA 19104, 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 properties of the assembled system through analysis of individual components before assembly. This work introduces such a modular technique in the context of testing. Assume-guarantee testing relies on the (automated) decomposition of key system-level requirements into local component requirements at design time. Developers can verify the local requirements by checking components in isolation; failed checks may indicate violations of system requirements, while valid traces from different components compose via the assume-guarantee proof rule to potentially provide system coverage. These local requirements also form the foundation of a technique for efficient predictive testing of assembled systems: given a correct system run, this technique can predict violations by alternative system runs without constructing those 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 software production cost. Developers could use such techniques to avoid expensive verification of assembled systems, 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 nontrivial. 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, 13, 15, 17]. 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. Extensions that use an assumption for each component in the system also exist. Our previous work developed techniques that *automatically* generate assumptions for performing assume-guarantee model checking at the design level [2, 5, 9], ameliorating the often difficult challenge of finding an appropriate assumption.

While design verification is important, it is also necessary to verify that an 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 [10]; with this methodology, it is possible to verify source code one component at a time. Hence, this technique has the 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 languages commonly used in industry. This work explores the benefits of assume-guarantee reasoning for testing, which is still the predominant industrial verification technique. We have developed *assume-guarantee testing*, which reuses properties, assumptions, and proof rules from design-level assume-guarantee verification to enhance both unit testing and whole-system testing. The contributions of assume-guarantee testing are as follows:

1. During unit testing, assume-guarantee testing has the potential to obtain *system coverage* and detect *system-level errors*. Our approach applies assume-guarantee reasoning to component test traces, using assumptions as environ-

ments to drive individual components. This process provides guarantees on trace *compositions* that are analogous to the guarantees obtained by assume-guarantee model checking. Hence, the technique can infer that a (potentially large) set of system traces satisfies a global property by checking traces of components in isolation against assume-guarantee pairs. Moreover, component test traces that fail their assume-guarantee premises may uncover *system-level* violations. Assumptions restrict the context of the components, thus reducing the number of false positives obtained by verification (*i.e.*, errors that will never exhibit themselves in the context of the particular system in which the component will be introduced). As a result, the likelihood that a failed local check corresponds to a system-level error is higher. Early error detection is desirable, as it is well established that errors discovered earlier in the development phase are easier and cheaper to fix.

2. During whole-system testing, assume-guarantee testing has the potential to efficiently detect bugs and provide coverage. In this context, our approach projects system traces onto individual components, and applies assume-guarantee reasoning to the projections. This technique is an efficient means of *predictive testing.* Predictive testing detects the existence of bad traces from good traces [19]. It exploits the insight that one can reorder independent events from a trace to obtain different legal traces. Typically, predictive testing techniques discover these alternative traces by composing independent events in different orders. Our technique uses assume-guarantee reasoning to obtain results about the alternative interleavings *without* explicitly exploring them, and thus is potentially more efficient.

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; monolithic model checking would therefore not have detected it.

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, this work uses Labeled Transition Systems (LTSs) to model the behavior of communicating components. Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment. An LTS M is a quadruple $\langle Q, \alpha M, \delta, q0\rangle$ where:

- Q is a non-empty finite set of states
- $\alpha M \subseteq \mathcal{A}ct$ 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
- $q0 \in Q$ is the initial state

Let $M = \langle Q, \alpha M, \delta, q0\rangle$ and $M' = \langle Q', \alpha M', \delta', q0'\rangle$. We say that M *transits* into M' with action a, denoted $M \xrightarrow{a} M'$, if and only if $(q0, a, q0') \in \delta$ 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 $\Sigma \subseteq \mathcal{A}ct$, we use $t{\restriction}\Sigma$ 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 $t = \langle a_1, a_2, \ldots, a_n\rangle$ be a finite trace of some LTS M. We use $[t]$ to denote the LTS $M_t = \langle Q, \alpha M, \delta, q0\rangle$ with $Q = \{q_0, q_1, \ldots, q_n\}$, and $\delta = \{(q_{i-1}, a_i, q_i)\}$, where $1 \leq i \leq n$.

Parallel Composition. The parallel composition operator $\|$ 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 $M_1 = \langle Q_1, \alpha M_1, \delta_1, q0_1\rangle$ and $M_2 = \langle Q_2, \alpha M_2, \delta_2, q0_2\rangle$ be two LTSs. Then $M_1 \| M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q0\rangle$, where $Q = Q_1 \times Q_2$, $q0 = (q0_1, q0_2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows, where a is either an observable action or τ (note that commutativity implies the symmetric rules):

$$\frac{M_1 \xrightarrow{a} M_1',\ a \notin \alpha M_2}{M_1 \| M_2 \xrightarrow{a} M_1' \| M_2}$$

$$\frac{M_1 \xrightarrow{a} M_1',\ M_2 \xrightarrow{a} M_2',\ a \neq \tau}{M_1 \| M_2 \xrightarrow{a} M_1' \| M_2'}$$

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

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 [17]. 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, the user expresses all of A, M, P 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 $M_1 \| M_2 \models P$ without composing M_1 with M_2. For this purpose, the simplest proof rule 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 $\langle true\rangle\ M_1 \| M_2\ \langle P\rangle$ also holds. Note that for this rule to be useful, 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. Unfortunately, it is often difficult to find such an assumption.

Our previous work developed frameworks that compute assumptions automatically for finite-state models and safety properties expressed as LTSs. More specifically, Giannakopoulou et al. [9] 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. Barringer et al. [2] and Cobleigh et al. [5] use a learning algorithm 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. This work assumes a top-down software development process, where one creates and debugs design models and then uses these models 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, developers can use the feedback provided by the verification framework to correct the models). 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 design-level verification establishes the property, it is necessary to verify that the property holds at the implementation level, *i.e.* that $C_1 \parallel C_2 \models P$. This work assumes that each component implements one of the design models, *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 the proof rule guarantees 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 for a trace to be legal, *in* must occur in the trace before any occurrence of *out*. Figure 2 also shows the assumption A that design-level analysis of $M_1 \parallel M_2$ generates (see Section 2). Note that although $M_1 \parallel M_2 \models P$, $C_1 \parallel C_2$ does not. Testing C_1 and C_2 in isolation may produce the traces t_1 and t_2 (respectively) that Figure 3 (left) shows. 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 the developers do not use assume-guarantee testing, but rather test the assembled system (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 the developers to mistakenly believe that the system is correct. They may even achieve some coverage criterion without detecting the bug, as discussed later in this section.

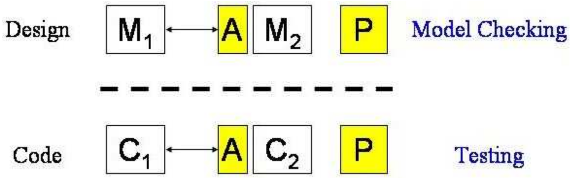

Figure 1: Design and code level analysis

In summary, assume-guarantee testing 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 testing of assembled systems. When the system is assembled, the testing framework uses assume-guarantee reasoning to conduct analysis that can efficiently predict, based on correct system runs, violations by alternative system runs. We discuss both flavors of assume-guarantee testing in more detail below.

3.1 Assume-Guarantee Component Testing

The first step in assume-guarantee component 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. The next step is to execute C_1 in U_A and C_2 in U to produce sets of traces T_1 and T_2 respectively. The technique then performs assume-guarantee reasoning, checking each trace $t_1 \in T_1$ against P and each trace $t_2 \in T_2$ 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, which the developers can then correct. If all these tests succeed, then the assume-guarantee rule implies that $[t_1]|||[t_2] \models P$, for all $t_1 \in T_1$ and $t_2 \in T_2$.

Using this approach, one can check system correctness through local tests of components. It is possible to perform assume-guarantee testing as soon as each component becomes "code complete", and *before* assembling an executable system or even implementing other components. 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

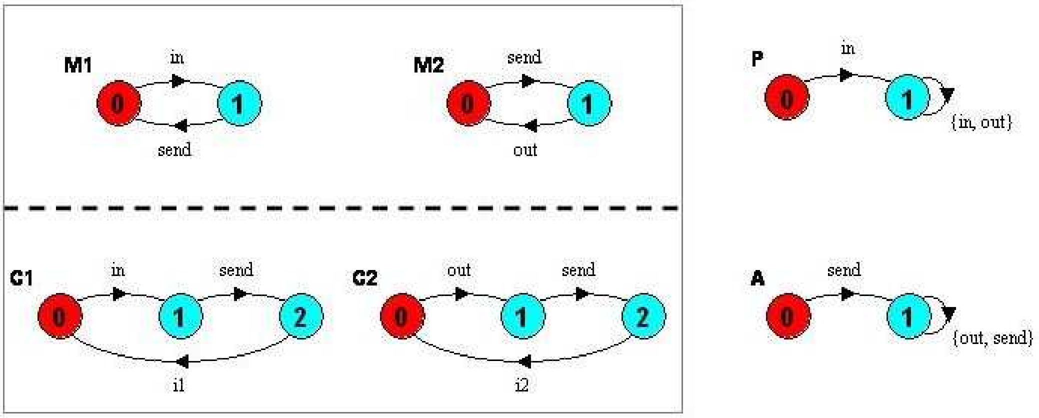

Figure 2: Assume-guarantee testing

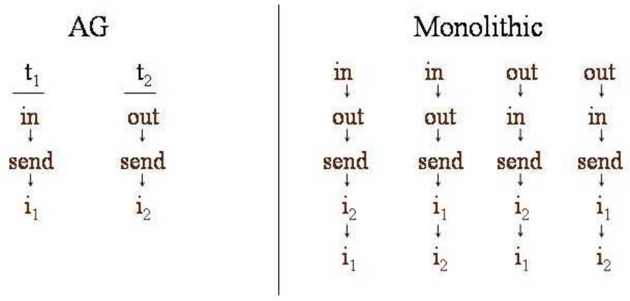

Figure 3: Discovering bugs with fewer tests

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 generated assumptions are much smaller than the component models. Therefore, it is more efficient to check the assumptions than to check refinement directly. 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, software quality assurance engineers and researchers on software testing have traditionally associated the notion of *coverage* 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 a system's implementation. 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 potential 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. The question of what benefits assume-guarantee reasoning can provide in such a context is open research.

Discussion. As stated above, our hope is that by checking individual traces of components, the technique covers 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 give the empty trace on composition. 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, assume-guarantee testing faces the question of producing compatible traces during component testing. One potential way to guarantee that T_1 and T_2 contain compatible traces is 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 on Assembled Systems

Assume-guarantee testing can also be a mechanism for predictive testing of component assemblies. Assume-guarantee testing for predictive analysis has the following steps:

- Obtain a system trace t (by running $C_1||C_2$).
- Project the trace on the alphabets of each component; obtain $t_1 = t{\upharpoonright}\alpha C_1$ and $t_2 = t{\upharpoonright}\alpha C_2$.
- 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.

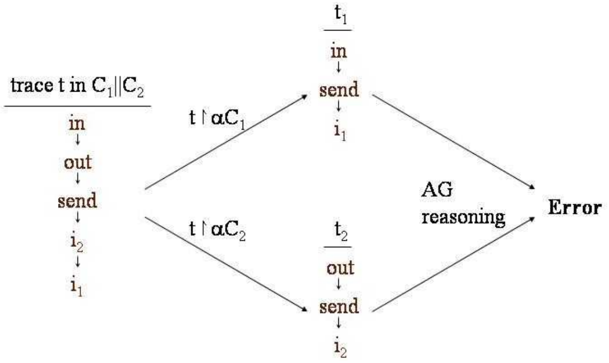

Figure 4: Predictive analysis

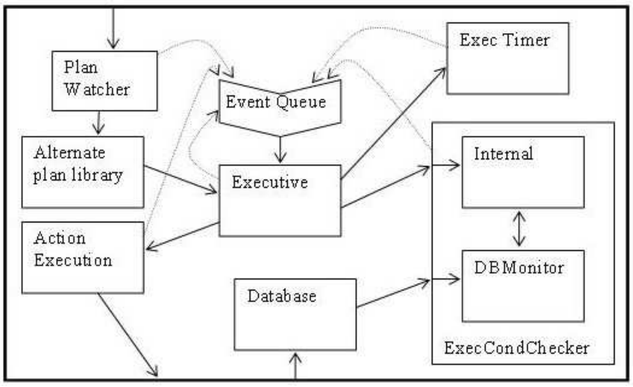

Figure 5: The Executive of the K9 Mars Rover

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 the methodology has been able to show that $C_1||C_2$ violates the property. 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) as long as the assumption is small; in our experience, this is often the case [9].

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, it may not be practical to generate a new assumption for each trace; we plan to experiment with this approach in the future.

Discussion. It is desirable 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. However, to gain the full benefits of assume-guarantee testing in this context, trace generation should take into account the results of predictive analysis. For example, suppose that trace generation produces a trace t, projected onto t_1 and t_2. Assume-guarantee testing proves that $[t_1]|||[t_2] \models P$. Further trace generation should avoid traces in $[t_1]|||[t_2]$ since these are covered by the assume-guarantee checking of t_1 and t_2. Again, one possible way to ensure avoidance of such redundant traces is to use 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. Test input generation techniques could also be useful for this purpose. This topic is a subject of future work.

4. EXPERIENCE

Our case study is the planetary rover controller K9, and in particular its executive subsystem, developed at NASA Ames Research Center. We performed this study in the context of an ongoing collaboration with the developers of the rover, in which we have performed verification *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 implementation is 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*, and an *ActionExecution* thread that is responsible for issuing the commands (actions) to the rover. The communication 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

We previously developed detailed design models for the executive subsystem [9]. 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). We implemented each environment as

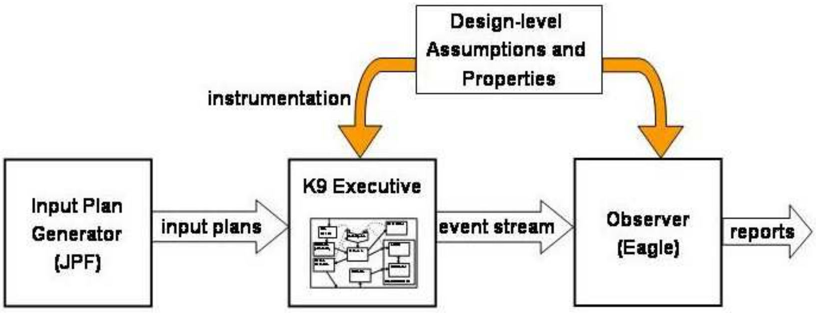

Figure 6: Testing Environment

a thread running a state machine (the respective design-level assumption) that executes in an infinite loop. In each iteration of the loop, the environment makes a random choice to perform an "active" event (such as calling a component function) that is enabled in the current state; the state machine then makes the appropriate transition. 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). The environment implementations also provide stubs for the external functions that the component under testing calls; when called, these functions cause state machine transitions.

The methodology 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, the user instruments the program to emit events that provide a trace of the running system. Eagle then checks 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 also (automatically) translated these assumptions and properties into Eagle monitors.

Note that in order to run the executive system (or its components), the user needs 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 [21]. 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 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 verification techniques can analyze the individual components of large systems 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.

Our previous work [10] presented a technique for using design-level assumptions for compositional analysis of source code. That work used model checking (Java PathFinder [20]), while the focus here is on testing. Dingel [7] also uses model checking (the VeriSoft state-less model checker [11]) for performing assume-guarantee verification for C/C++ components. However, the burden of generating assumptions is on the user.

Our work is also related to specification-based testing, a widely-researched topic. For example, Jagadeesan et al. [14] and Raymond et al. [18] use formal specifications for the generation of test inputs and oracles. These works generate test inputs from constraints (or assumptions) on the environment of a software component and test oracles from the guarantees of the component under test. The AsmLT Test Generator [12] translates Abstract State Machine Language (AsmL) specifications into finite state machines (FSMs) and different traversals of the FSMs are used to construct test inputs. We plan to investigate the use of different traversal techniques for test input generation from assumptions and properties (which are in essence FSMs). None of the above-described approaches address predictive analysis.

Sen et al. [19] have also explored predictive runtime analysis of multithreaded programs. 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 this work proposes using assume-guarantee reasoning to give similar guarantees without having to explore the composition of the projected traces.

Levy et al. [16] use assume-guarantee reasoning in the context of runtime monitoring. Unlike our work, which aims at improving testing, the goal of their work 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. During unit testing, our approach uses design-level assumptions as environments for individual components and checks generated traces against premises of an assume-guarantee proof rule; the assumptions restrict the context in which the components operate, making it more likely that failed checks correspond to system-level errors. During testing of component assemblies, the technique uses assume-guarantee reasoning on component projections of a system trace, providing results on alternative system traces. 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, quantifying its benefits is a difficult task. 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 interesting avenue for future research in this area is the use of the models as a coverage metric.

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 *International Conference on Computer-Aided Verification*, June–July, 1998.

[2] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *International Workshop on Specification and Verification of Component-Based Systems*, Sept. 2003.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *International Conference on Verification, Model Checking and Abstract Interpretation*, Jan. 2004.

[4] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional model checking. In *Symposium on Logic in Computer Science*, June 1989.

[5] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr. 2003.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *Symposium on the Foundations of Software Engineering*, Sept. 2001.

[7] J. Dingel. Computer assisted assume guarantee reasoning with VeriSoft. In *International Conference on Software Engineering*, May 2003.

[8] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *European Symposium on Programming*, Apr. 2002.

[9] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *International Conference on Automated Software Engineering*, Sept. 2002.

[10] D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *International Conference on Software Engineering*, May 2004.

[11] P. Godefroid. Model checking for programming languages using VeriSoft. In *Symposium on Principles of Programming Languages*, Jan. 1997.

[12] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *International Symposium on Software Testing and Analysis*, July 2002.

[13] O. Grumberg and D. E. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, Aug. 1991.

[14] 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*, May 1997.

[15] C. B. Jones. Specification and design of (parallel) programs. In *IFIP World Computer Congress*, Sept. 1983.

[16] J. Levy, H. Saidi, and T. E. Uribe. Combining monitors for run-time system verification. *Electronic Notes in Theoretical Computer Science*, 70(4), Dec. 2002.

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

[18] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Real-Time Systems Symposium*, Dec. 1998.

[19] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Conference on Formal Methods for Open Object-Based Distributed Systems*, 2005.

[20] W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *International Conference on Automated Software Engineering*, Sept. 2000.

[21] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis*, July 2004.