Breaking Up is Hard to Do: An Evaluation of Automated Assume-Guarantee Reasoning

JAMIESON M. COBLEIGH, GEORGE S. AVRUNIN,
and LORI A. CLARKE
University of Massachusetts Amherst

7

Finite-state verification techniques are often hampered by the state-explosion problem. One proposed approach for addressing this problem is assume-guarantee reasoning, where a system under analysis is partitioned into subsystems and these subsystems are analyzed individually. By composing the results of these analyses, it can be determined whether or not the system satisfies a property. Because each subsystem is smaller than the whole system, analyzing each subsystem individually may reduce the overall cost of verification. Often the behavior of a subsystem is dependent on the subsystems with which it interacts, and thus it is usually necessary to provide assumptions about the environment in which a subsystem executes. Because developing assumptions has been a difficult manual task, the evaluation of assume-guarantee reasoning has been limited. Using recent advances for automatically generating assumptions, we undertook a study to determine if assume-guarantee reasoning provides an advantage over monolithic verification. In this study, we considered *all* two-way decompositions for a set of systems and properties, using two different verifiers, FLAVERS and LTSA. By increasing the number of repeated tasks in these systems, we evaluated the decompositions as they were scaled. We found that in only a few cases can assume-guarantee reasoning verify properties on larger systems than monolithic verification can, and in these cases the systems that can be analyzed are only a few sizes larger. Although these results are discouraging, they provide insight about research directions that should be pursued and highlight the importance of experimental evaluation in this area.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

This is a revised and extended version of a paper presented at ISSTA 2006 in Portland, Maine.
This research was partially supported by the National Science Foundation under grants CCR-0205575, CCF-0427071, and CCF-0541035, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement DAAD190110564, and by the U.S. Department of Defense/Army Research Office under agreement DAAD190310133. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Army Research Office, or the U.S. Department of Defense/Army Research Office.
J. M. Cobleigh is currently affiliated with The MathWorks, 3 Apple Hill Drive, Natick, MA 01760.
Authors' addresses: J. M. Cobleigh, G. S. Avrunin, L. A. Clarke, Department of Computer Science, University of Massachusetts, 140 Governor's Drive, Amherst, MA 01003-9264; email: {jcobleig,avrunin,clarke}@cs.umass.edu.

General Terms: Verification, Experimentation

Additional Key Words and Phrases: Assume-guarantee reasoning

ACM Reference Format:

Cobleigh, J. M., Avrunin, G. S., and Clarke, L. A. 2008. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. ACM Trans. Softw. Engin. Method. 17, 2, Article 7 (April 2008), 52 pages. DOI = 10.1145/1348250.1348253 http://doi.acm.org/10.1145/1348250.1348253

1. INTRODUCTION

Software systems are taking on an increasingly important role in society and are being used in critical applications where their failure could result in human casualties or substantial economic loss. Thus, it is important to validate such software systems to ensure their quality. This task is becoming more difficult, however, as software systems continue to increase both in size and in complexity. There are many techniques that can be used to validate software systems, one of which is *finite-state verification* (FSV). FSV techniques work by analyzing a finite model of a system to ensure that it satisfies a property specifying a desired system behavior. Since FSV techniques examine all paths through the system model, they can be used to determine whether or not the property being verified is violated. If the property is violated, FSV techniques usually provide a counterexample, a path through the model that reveals this violation. FSV techniques, however, are limited in the size of the system that they can evaluate, since the cost of verification can be exponential in the size of the system being verified; this is refered to as the state-explosion problem.

Compositional analysis techniques use a divide-and-conquer approach to verification in an attempt to reduce the effects of the state-explosion problem. One of the most frequently advocated compositional analysis techniques is *assume-guarantee reasoning* [Jones 1983; Pnueli 1984] where a system under analysis is decomposed into subsystems and these subsystems are analyzed individually. By composing the results of these analyses, it can be determined whether or not the whole system satisfies the original property. By individually analyzing the subsystems, each of which is smaller than the whole system, the effects of the state-explosion problem may be reduced. Often the behavior of a subsystem is dependent on the subsystems with which it interacts, and thus it is usually necessary to provide assumptions about the environment in which a subsystem executes to verify properties of that subsystem.

In assume-guarantee reasoning, a verification problem is represented as a triple, $\langle A \rangle\ S\ \langle P \rangle$, where:

(1) S is the subsystem being analyzed,
(2) P is the property to be verified, and
(3) A is an assumption about the environment in which S is used.

Note that although this notation resembles a Hoare triple [Hoare 1969], A is not a precondition and P is not a postcondition. Instead, A is a constraint on the

behavior of S. If S, as constrained by A, satisfies P, then the formula $\langle A \rangle\ S\ \langle P \rangle$ is true.

Consider a system that is decomposed into two subsystems, S_1 and S_2. To verify that a property P holds on the system composed of S_1 and S_2 running in parallel, denoted $S_1 \parallel S_2$, the following is the simplest assume-guarantee rule that can be used:

$$\begin{array}{ll} \text{Premise 1:} & \langle A \rangle\ S_1\ \langle P \rangle \\ \text{Premise 2:} & \langle \mathit{true} \rangle\ S_2\ \langle A \rangle \\ \hline & \langle \mathit{true} \rangle\ S_1 \parallel S_2\ \langle P \rangle \end{array}$$

This rule states that if under assumption A subsystem S_1 satisfies property P and subsystem S_2 satisfies an assumption A, then the system $S_1 \parallel S_2$ satisfies property P. This allows a property to be verified on $S_1 \parallel S_2$ without ever having to examine a monolithic model for the entire system.

There are several issues that make using this assume-guarantee rule difficult. First, if the system under analysis is made up of more than two subsystems, which is often the case, then S_1 and S_2 may each need to be made up of several of these subsystems. How this decomposition is done can have a significant impact on the time and memory needed for verification, but it is not clear how to select an effective decomposition. In fact, we have found that the memory usage between two different decompositions can vary by over an order of magnitude. Second, once a decomposition is selected, it can be difficult to manually find an assumption A that can be used to complete an assume-guarantee proof because the assumption must:

(1) be strong enough to sufficiently constrain the behavior of S_1 so that $\langle A \rangle\ S_1\ \langle P \rangle$ holds, and
(2) be weak enough so that $\langle \mathit{true} \rangle\ S_2\ \langle A \rangle$ holds.

Because selecting a decomposition and developing an assumption are difficult tasks, it had not been practical previously to undertake an empirical evaluation of assume-guarantee reasoning, although several case studies have been reported [e.g., Henzinger et al. 1998; McMillan 1998; Fournet et al. 2004].

Recent work on automatically computing assumptions for assume-guarantee reasoning [Giannakopoulou et al. 2002; Barringer et al. 2003; Cobleigh et al. 2003; Henzinger et al. 2003; Alur et al. 2005; Chaki et al. 2005] eliminates one of the obstacles to empirical evaluation by making it feasible to examine a large number of decompositions without having to manually produce a suitable assumption for each one. Using one algorithm that learns assumptions [Cobleigh et al. 2003], we undertook a study to evaluate the effectiveness of assume-guarantee reasoning.

We first conducted a preliminary study to gain insight into how to best decompose systems and to learn what kind of savings could be expected from assume-guarantee reasoning. We began by using the FLAVERS FSV tool [Dwyer et al. 2004] to verify properties of several systems written in Ada, but the results of these experiments were not as promising as the results seen in Cobleigh et al. [2003], which applied LTSA [Magee and Kramer 1999], another FSV tool, to a single system derived from a C++ program. Although

FLAVERS and LTSA use different models and verification methods, this discrepancy was surprising to us. As a result, we also translated the Ada systems into the input language of LTSA, to see if our choice of tool affected our results. Using both FSV tools, we selected several decompositions for each example at the smallest reasonable system size based on our understanding of the system, the property, and assume-guarantee reasoning. We expected that in most cases assume-guarantee reasoning would save memory over monolithic verification. In this study, we were surprised to discover that in over half of the subjects we verified, the decompositions we selected did not use less memory than monolithic verification, regardless of the verifier used.

Based on the results of this preliminary study, we undertook the more comprehensive study that is reported in this article. We started by looking at the smallest size of each of our example systems. For each system, property, and verifier, we found the best decomposition, in the sense that assume-guarantee reasoning explores the fewest states, by examining all of the ways to decompose that system into S_1 and S_2. Then, because examining all decompositions at larger system sizes quickly becomes infeasible due to the explosion in the number of decompositions that need to be considered and the increased cost for evaluating each decomposition, we generalized the best decompositions found for smaller system sizes and used those generalized decompositions when applying assume-guarantee reasoning on larger system sizes. To evaluate these generalized decompositions we tried to explore all two-way decompositions for a few larger system sizes, although we were not always able to find the best decomposition because of the time required. In total we examined over 43,500 two-way decompositions, which used over 1.54 years of CPU time.

The results of our experiments are not very encouraging and raise concerns about the effectiveness of assume-guarantee reasoning. For the vast majority of decompositions, more states are explored using assume-guarantee reasoning than are explored using monolithic verification. If we restrict our attention to just the best decomposition for each property, we found that in about half of these cases our automated assume-guarantee reasoning technique explores fewer states than monolithic verification for the smallest system size. When we used generalized decompositions to scale the systems, assume-guarantee reasoning often explores fewer states than monolithic verification. This memory savings, however, is rarely enough to increase the size of the systems that can be verified beyond what can be done with monolithic verification. Although these results are discouraging, they provide insight about research directions that should be pursued and highlight the importance of experimental evaluation for this area.

Section 2 provides some background information about the finite-state verifiers and the automated assume-guarantee algorithm we used. Section 3 describes the automated assumption generation algorithms for FLAVERS and for LTSA. Section 4 describes our experimental methodology and results. Section 5 discusses related work. We end by presenting our conclusions and discussing future work.

2. BACKGROUND

This section gives a brief description of FLAVERS and LTSA, the two FSV tools used in our experiments. It also briefly describes the L* algorithm and how it can be used to automatically learn assumptions for use in assume-guarantee reasoning.

2.1 FLAVERS

FLAVERS (**FL**ow **A**nalysis for **VER**ification of **S**ystems) [Dwyer et al. 2004] is an FSV tool that can prove user-specified properties of sequential and concurrent systems. These properties need to be expressed as sequences of events that should (or should not) happen on any execution of the system. A property can be expressed in a number of different notations, but must be translatable into a Finite State Automaton (FSA). The model FLAVERS uses to represent a system is based on annotated Control Flow Graphs (CFGs). Annotations are placed on nodes of the CFGs to represent events that occur during execution of the actions associated with a node. Since a CFG represents the potential control flow of a sequential system, this representation is not sufficient for modeling a concurrent system. FLAVERS uses a *Trace Flow Graph* (TFG) to represent a concurrent system. The TFG consists of a collection of CFGs, with each CFG representing a task in the system, and with additional nodes and edges to represent potential intertask control flow. Typically a CFG, and thus a TFG, overapproximates the sequences of events that can occur when executing a system.

FLAVERS uses an efficient state-propagation algorithm [Olender and Osterweil 1992; Dwyer et al. 2004] to determine whether all potential behaviors of the system being analyzed are consistent with the property being verified. FLAVERS analyses are conservative, meaning FLAVERS will only report that the property holds when the property holds for all TFG paths. If FLAVERS reports that the property does not hold, this may be because at least one of the violating traces through the TFG corresponds to an actual execution of the system, and thus there is an error in the system, in the property, or in both. Alternatively, the property may only be violated on infeasible paths, which are paths that do not correspond to any possible execution of the system but are an artifact of the imprecision of the model. The analyst can then introduce *constraints*, also represented as FSAs, to improve the precision of the model and thereby eliminate some infeasible paths from consideration. An analyst might need to iteratively add constraints and observe the analysis results several times before determining whether or not a property holds. Constraints give analysts some control over the analysis process by letting them determine exactly which parts of a system need to be modeled to prove a property.

The FLAVERS state-propagation algorithm has worst-case complexity that is $O(N^2 \cdot P \cdot C_1 \cdots C_k)$, where N is the number of nodes in the TFG, P is the number of states in the property, and C_i is the number of states in the ith constraint. Experimental evidence shows that the performance of FLAVERS is often sub-cubic in the size of the system [Dwyer et al. 2004] and that the

performance of FLAVERS is good when compared to other finite-state verifiers [Avrunin et al. 1999, 2000].

2.2 LTSA

LTSA (**L**abeled **T**ransition **S**ystems **A**nalyzer) [Magee and Kramer 1999] is another FSV tool that can prove user-specified properties of sequential and concurrent systems. LTSA can check both safety and liveness properties, but the assume-guarantee algorithm we used could only handle safety properties. Safety properties in LTSA are specified as FSAs where every state except one is an accepting state. This single nonaccepting state must be a trap state, meaning it has no transitions to other states. This type of FSA corresponds to the set of prefix-closed regular languages, those languages where every prefix of every string in the language is also in the language. LTSA uses *Labeled Transition Systems* (LTSs), which resemble FSAs, for modeling the components of a system. In LTSA, LTSs are written in *Finite State Process* (FSP), a process-algebra style notation [Magee and Kramer 1999].

Unlike FLAVERS, in which the nodes of the model are labeled with the events of interest, in LTSA, the edges (or transitions) of the LTSs are labeled with the events of interest. To build a model of the entire system, individual LTSs are combined using the parallel composition operator ($\|$). The parallel composition operator is a commutative and associative operator that combines the behavior of two LTSs by synchronizing the events common to both and interleaving the remaining events. LTSA uses reachability analysis to verify a property and has a worst-case complexity that is $O(N_1 \cdots N_k \cdot P)$ where N_i is the number of states in the ith LTS in the system and P is the number of states in the property.

2.3 Using the L* Algorithm to Automate Assume-Guarantee Reasoning

The L* algorithm was originally developed by Angluin [1987] and later improved by Rivest and Schapire [1993]. In our work, we used Rivest and Schapire's version of the L* algorithm. The L* algorithm learns an FSA for an unknown regular language over an alphabet Σ by interacting with a *minimally adequate teacher*, henceforth referred to as a *teacher*. The L* algorithm poses two kinds of questions to the teacher, queries and conjectures, where the questions posed are based on the answers received to previously asked questions. For assume-guarantee reasoning, the L* algorithm is used to learn an FSA that recognizes the language of the weakest possible assumption under which S_1 satisfies P [Giannakopoulou et al. 2002]. The weakest possible assumption is unique, assuming it has been minimized to ensure a canonical representation [Giannakopoulou et al. 2002]. For an assumption A, let $\mathcal{L}(A)$ denote the language of that assumption. The weakest possible assumption A_w is an assumption for which

(1) $\langle A_w \rangle\ S_1\ \langle P \rangle$ is true and
(2) $\forall A$ such that $\mathcal{L}(A) \supset \mathcal{L}(A_w)$, $\langle A \rangle\ S_1\ \langle P \rangle$ is false.

We refer the reader to Rivest and Schapire [1993] for a description of the L* algorithm itself and here only describe how queries and conjectures are answered to allow the L* algorithm to learn an assumption A that can be used in the simple assume-guarantee rule given earlier.

2.3.1 *Answering Queries.* A *query* consists of a sequence of events from Σ^* where the teacher must return *true* if the string is in the language being learned and *false* otherwise. In answering queries for assume-guarantee reasoning, the focus is on Premise 1, $\langle A \rangle\ S_1\ \langle P \rangle$. To answer a query, the model of S_1 is examined to determine if the given event sequence results in a violation of the property P. If it does, then the assumption needed to make $\langle A \rangle\ S_1\ \langle P \rangle$ true should not allow the event sequence in the query, and thus *false* will be returned to the L* algorithm. Otherwise, the event sequence is permissible and *true* will be returned to the L* algorithm.

2.3.2 *Answering Conjectures.* A *conjecture* consists of an FSA that the L* algorithm believes will recognize the language being learned. The teacher must return *true* if the conjecture is correct. Otherwise, the teacher must return *false* and a counterexample, a string in Σ^* that is in the symmetric difference of the language of the conjectured automaton and the language being learned. In the context of assume-guarantee reasoning, the conjectured FSA is a candidate assumption that may be able to be used to complete an assume-guarantee proof. Thus, conjectures are answered by determining if the conjectured assumption makes the two premises of the assume-guarantee proof rule true.

To answer a conjecture, the candidate assumption, A, is first checked to see if it satisfies Premise 1. To check this, the model of S_1, as constrained by the assumption A, is verified. If this verification reports that P does not hold, then the counterexample returned represents an event sequence permitted by A that can cause S_1 to violate P. Thus, the conjecture is incorrect and the counterexample is returned to the L* algorithm. If the verification reports that the property does hold, then A is good enough to satisfy Premise 1 and Premise 2 can be checked.

Premise 2 states that $\langle true \rangle\ S_2\ \langle A \rangle$ should be true. To check this, the model for S_2 is verified to see if it satisfies A. If this verification reports that A holds, then both Premise 1 and Premise 2 are true, so it can be concluded that P holds on $S_1 \parallel S_2$ and the automated assume-guarantee reasoning algorithm can stop. If this verification reports that A does not hold, then the resulting counterexample is examined to determine what should be done next.

The first thing considered is to make a query to see if the event sequence of the counterexample leads to a violation of the property P on S_1. If a property violation results, then the counterexample is a behavior that occurs in S_2 that can result in a property violation when S_2 interacts with S_1, so it can be concluded that P does not hold on $S_1 \parallel S_2$ and the automated assume-guarantee reasoning algorithm can stop. If a property violation does not occur, then the counterexample is a behavior that occurs in S_2 that will not result in a property violation when S_2 interacts with S_1, and thus A is saying that certain behaviors of S_2 are illegal when they actually are legal. Thus, the assumption

A is incorrect and the counterexample is then returned to the L* algorithm in response to the conjecture.

Note that Premise 1 is checked before Premise 2 because doing so allows the algorithm to answer the conjecture immediately if Premise 1 is false. If Premise 2 were checked first and the result were false, a query would still need to be made before the conjecture could be answered. Thus, we chose to check Premise 1 in an attempt to make the algorithm more efficient.

2.3.3 *Complexity and Correctness.* This approach to assume-guarantee reasoning terminates and correctly determines whether or not $S_1 \parallel S_2$ satisfies P [Cobleigh et al. 2003].

Using Rivest and Schapire's version of the L* algorithm, $l - 1$ conjectures and $O(kl^2 + l \log m)$ queries are needed in the worst case, where k is the size of the alphabet of the FSA being learned, l is the number of states in the minimal deterministic FSA that recognizes the language being learned, and m is the length of the longest counterexample returned when a conjecture is made. Since we are using an FSV tool to generate counterexamples, we are guaranteed that there will be a finite number of states explored during analysis. The maximum number of states explored provides an upper bound on the length of the longest counterexample. In our experience, counterexamples are significantly shorter than this worst-case upper bound.

Although this approach to assume-guarantee reasoning is correct and will terminate, it is not guaranteed to save memory over monolithic verification. When the L* algorithm is learning an assumption A to make Premise 1 true, it might learn an assumption that has fewer states than S_2 but that allows behaviors that do not occur in S_2. As a result, the composition of S_1 and A could have more behaviors than the composition of S_1 and S_2. This would likely lead to the size of $S_1 \parallel S_2$ being larger then the size of $S_1 \parallel A$, and result in the verification of Premise 1 using more memory than monolithic verification.

Additionally, while the learned assumption is expected to be smaller than S_2, it is often larger than the property P. As a result, checking Premise 2 may use more memory than monolithic verification.

Finally, this approach is also not guaranteed to save time since there is additional overhead for learning an assumption.

3. IMPLEMENTING THE TEACHERS

The teacher, as described at a high level in Section 2.3, works for both FLAVERS and LTSA. The LTSA teacher, which was used as the basis for the teacher for FLAVERS, is described in detail in Cobleigh et al. [2003]. Differences in the models used by FLAVERS and LTSA, however, necessitate differences in the implementations of their teachers. In this section, we focus on the most significant difference between the two teachers, which is how the models for S_1 and S_2 are built. A detailed description of the teacher for FLAVERS is given in the Appendix.

The systems used in our experiments are written in Ada, which uses rendezvous for intertask communication. Consider a system made up of two tasks,

T_1 and T_2, and let $S_1 = \{T_1\}$ and $S_2 = \{T_2\}$. Suppose that T_1 and T_2 communicate via a rendezvous "r" that is accepted once by T_1 and called once by T_2.

In LTSA, this rendezvous is represented by two transitions, one in the LTS for T_1 and one in the LTS for T_2, that are labeled with the same event. In the LTS model, when an LTS is in a given state, only events on enabled transitions can occur. A transition with event e is enabled if:

(1) e is τ, where τ represents an action that is invisible to the environment of the LTS,
(2) e is not in the alphabet of any other LTSs, or
(3) e occurs in other LTSs and e is on a transition out of the current state of every other LTS that has e in its alphabet.

Thus, when building the model for S_1, the event that corresponds to the rendezvous "r" would always be enabled since there are no other LTSs in S_1 that have "r" in their alphabets. Although T_2 also has "r" in its alphabet, T_2 is in S_2, not S_1.

In FLAVERS, this rendezvous is represented by two nodes, one in the CFG for T_1, annotated as accepting the rendezvous "r," and one in the CFG for T_2, annotated as calling the rendezvous "r." For monolithic verification, when the TFG is built from CFGs for T_1 and T_2, these two nodes are replaced by a single node that represents the occurrence of the rendezvous "r." If a model for S_1 is built using just the CFG for T_1, FLAVERS removes the node that corresponds to the accept of "r" because there is no node in the collection of CFGs that makes up S_1 that calls "r." As a result, any transitions in a property that would be taken when the rendezvous "r" occurs will never be taken because the rendezvous "r" would not be in the model. Thus, to accurately model S_1, we need to add an environment task that captures the interaction between tasks in S_1 and tasks in S_2. A similar environment needs to be built when constructing the model for S_2. Although tools exist for building environments for model checking software systems [Tkachuk et al. 2003], we do not use such tools since the environment generation problem in our context is not very complicated. For FLAVERS, we build an environment automatically after a simple analysis of the nodes in the CFGs of the system, as described in detail in the Appendix.

The way these environments are generated can affect the memory and time needed to check properties for assume-guarantee reasoning. Păsăreanu et al. [1999] looked at two approaches to environment generation. The first approach is to generate a universal environment and have its actions constrained by the assumption A. The second approach is to convert the assumption A into an environment that only allows the sequences of actions permitted by the assumption. Their experiments used SPIN and SMV and produced inconclusive results since neither approach for environment generation outperformed the other approach consistently. We used the first approach in our study. That is, we generated a universal environment and constrained it with an assumption. We chose this approach because it allows better reuse of artifacts, since the TFG does not need to be regenerated for each conjecture. In Section 4.7, we discuss

how this decision and others made in model and assumption construction might have influenced our experimental results.

4. METHODOLOGY AND RESULTS

There are several questions that could be asked to determine whether or not assume-guarantee reasoning provides an advantage over monolithic FSV. Since FSV techniques are more frequently limited by memory than by time, we primarily focused our study on the following two questions:

(1) Does assume-guarantee reasoning use less memory than monolithic verification?
(2) If assume-guarantee reasoning uses less memory than monolithic verification, is there enough savings to allow assume-guarantee reasoning to verify properties on larger systems than monolithic verification?

To evaluate the usefulness of this automated assume-guarantee reasoning technique, we tried to verify properties that were known to hold on a small set of scalable systems: the Chiron user interface system [Keller et al. 1991] (both the single and the multiple dispatcher versions as described in Avrunin et al. [1999]), the Gas Station problem [Helmbold and Luckham 1985], Peterson's mutual exclusion protocol [Peterson 1981], the Relay problem [Siegel and Avrunin 2002], and the Smokers problem [Patil 1971]. These systems were specified in Ada and use rendezvous for intertask communication. Except for Peterson's mutual exclusion protocol, which uses shared variables for intertask communication, these systems all have a client-server architecture where the server has an interface made up of a small number of rendezvous that may be called by the clients. The properties we checked on these systems are all safety properties that describe a legal (or illegal) sequence of events for each system and are given in Table I. Since our experiments examined the same properties on two different versions of the Chiron system, we use the term *subject* to refer to a property-system pair. Table I gives the subject number for each subject in the experiment. For the Chiron properties, two subject numbers are given, the one before the slash is for the subject in the single dispatcher system (henceforth referred to as "Chiron single") and the one after the slash is for the subject in the multiple dispatcher system (henceforth referred to as "Chiron multiple").

Each of the systems we used was scaled by creating more instances of one particular task, and the size of the system is measured by counting the number of occurrences of that task in the system. For the Chiron systems we counted the number of artists, for the Gas Station system we counted the number of customers, for the Peterson system we counted the number of tasks trying to gain access to the critical section, for the Relay system we counted the number of tasks accessing the shared variable, and for the Smokers system we counted the number of smokers. For each of the systems we looked at, we tried to verify properties starting at the smallest size and increasing the size up to a maximum size of 200. The somewhat arbitrary cutoff of 200 represents a significant size for the systems under consideration and provides substantial information about how the verification of that subject scales. While systems could be constructed

Table I. Description of the Properties

Subject Number(s)	Property Description
	Chiron
1 / 10	*artist1* never registers for *event1* if it is already registered for this event.
2 / 11	If *artist1* is registered for *event1* and the dispatcher receives *event1*, then the dispatcher will not accept another event before passing *event1* to *artist1*.
3 / 12	The dispatcher does not notify any artists of *event1* until it receives *event1*.
4 / 13	Having received *event1*, the dispatcher never notifies artists of *event2*.
5 / 14	If no artists are registered for *event1*, the dispatcher does not notify any artist of *event1*.
6 / 15	The dispatcher never gives *event1* to *artist1* if *artist1* is not registered for *event1*.
7 / 16	If *artist1* registers for *event1* before *artist2* does, then when the dispatcher receives *event1* it will first notify *artist1* and then *artist2* of this event.
8 / 17	The size of the list used to store the IDs of artists registered for *event1* never exceeds the number of artists.
9 / 18	The program does not terminate while there is an artist that is registered for an event.
	Gas Station
19	*customer1* and *customer2* cannot use *pump1* at the same time.
20	*customer1* repeatedly first starts pumping and then stops pumping.
21	*pump1* repeatedly lets a customer start pumping and then lets a customer stop pumping.
22	If *customer1* prepays on *pump1*, then *customer1* receives the change for *pump1*.
	Peterson
23	Two tasks cannot both be in the critical region at the same time.
	Relay
24	The shared variable is always set to 1 between any two times it is set to 0.
	Smokers
25	The correct smoker makes a cigarette after the supplier finishes putting out pieces.
26	*smoker1* assembles a cigarette after the supplier puts out the needed pieces.
27	Only one smoker can be making a cigarette at a time.
28	*smoker1* and *smoker2* cannot be making a cigarette at the same time.
29	The supplier never puts all of the pieces for a cigarette on the table at the same time.
30	After the supplier puts out items, one cigarette will be assembled.
31	Repeatedly, *piece1* is put on the table (by the supplier) and then *piece1* is picked up from the table (by a smoker).
32	The supplier and the smokers must first obtain a mutual exclusion lock before putting items on or taking items off the table and then they must release the lock.

for which some property holds at size 200 but not at size 201, for the systems we examined, verifying properties at size 200 provides high confidence that the properties will hold at even larger sizes.

For both FLAVERS and LTSA we considered a task to be an indivisible subsystem. Thus, a decomposition is an assignment of the tasks to either S_1 or S_2. Note that each scalable system we looked at had more than two subsystems (i.e., tasks), even at size 2.

Both FLAVERS and LTSA prove that a property holds by exploring all of the reachable states in an abstracted model of a system. On properties that do not hold, these two tools stop as soon as a property violation is found. As a result, their performance on properties that do not hold is more variable. Although using only properties that hold restricts the scope of our study, including properties that do not hold would have made it more difficult to meaningfully compare the performance of monolithic verification to assume-guarantee reasoning.

To determine the amount of memory used by monolithic verification, we counted the number of states explored during verification. While the artifacts created by the verifiers (e.g., TFGs and FSAs in FLAVERS, LTSs in LTSA) use memory, we did not count them when determining memory usage since the amount of memory needed to store them is usually small when compared to the amount of memory needed to store the states explored during verification. Similarly, to determine the amount of memory used by assume-guarantee reasoning, we looked at the maximum number of states explored by the teacher when answering a query or a conjecture of the L* algorithm. We consider one decomposition better than another decomposition if the maximum number of states explored when the teacher answers a query or conjecture using the first decomposition is smaller than the maximum number of states explored when the teacher answers a query or conjecture using the second decomposition.

For LTSA, INCA [Corbett and Avrunin 1995] was used to translate the Ada systems into FSAs, which are then easily translated into LTSs. There is one fewer Chiron property for LTSA than for FLAVERS. This property is shown to hold during model construction by INCA for LTSA, making verification using LTSA unnecessary. Because this property is applicable to each version of the Chiron system, there are two more subjects for FLAVERS than for LTSA, namely subjects 8 and 17.

We did not use the most recent version of LTSA, which is based on plugins [Chatley et al. 2004], because the plugin interface does not provide direct access to the LTSs. An implementation of the assumption-generation technique we used exists for the plugin version of LTSA [Giannakopoulou and Păsăreanu 2005], but verification takes significantly longer because all LTSs must be created by writing appropriate FSP, necessitating parsing the entire model for each query and conjecture, even for the parts of the model that do not change between different queries and conjectures.

We used the version of FLAVERS that directly accepts Ada systems. Since FLAVERS uses constraints to control the amount of precision in a verification, we used a minimal set of constraints when verifying properties. A minimal set of constraints is one that allows FLAVERS to verify that the property holds but

Table II. Number of Two-Way Decompositions Examined for Systems of Size 2

		FLAVERS		LTSA	
System	Decompositions	Properties	Total	Properties	Total
Chiron single	62	9	558	8	496
Chiron multiple	254	9	2,286	8	2,032
Gas Station	30	4	120	4	120
Peterson	6	1	6	1	6
Relay	6	1	6	1	6
Smokers	14	8	112	8	112
Total		**32**	**3,088**	**30**	**2,772**

also one where the removal of any constraint causes FLAVERS to report that the property may not hold.[1]

Tables with the details of our experimental results (e.g., the size of the learned assumptions and the number of states explored during verification) can be found in the Electronic Appendix for this paper. Also, all of our experimental subjects can be downloaded from:

`http://laser.cs.umass.edu/~jcobleig/breakingup-examples/`

4.1 Does Assume-Guarantee Reasoning Save Memory for Small System Sizes?

We began by looking at systems of size 2, the smallest reasonable size for all of the systems. For each subject in our study at size 2, we examined all two-way decompositions to find the best decomposition for that subject with respect to memory. For each systems at size 2, Table II lists the number of two-way decompositions[2] examined for each subject, the number of properties for each system, and the total number of decompositions examined for each system. Note that the Chiron multiple system has more decompositions than the Chiron single system because the changes made to the dispatcher resulted in a system with more tasks.

Figures 1 and 2 show, for FLAVERS and LTSA respectively, the amount of memory used by the best decomposition at size 2 normalized by dividing it by the amount of memory used by monolithic verification. For reference, a line at 1.0 has been drawn. Bars whose heights do not reach this line represent subjects on which the best decomposition is better than monolithic verification, while bars whose heights extend above this line represent subjects on which the best decomposition is worse than monolithic verification. Note that the subjects are ordered by their normalized memory usage and that the vertical axes are

[1]While these sets are minimal for each property, they may not be the smallest possible set of constraints with which FLAVERS can prove the property holds nor the best set with respect to the memory or time cost. While the worst-case complexity of FLAVERS increases with each constraint that is added, sometimes adding more constraints can improve the actual performance of FLAVERS. Since we did not consider all possible combinations of all possible constraints, we can not be certain that the selected minimal constraint set is either the smallest minimal set or the set that uses the least time or memory. On the other hand, a "reasonable" process that might be applied by analysts, described in Dwyer et al. [2004], was used to select the sets of constraints used in our study.

[2]Note that the number of two-way decompositions examined for each subject is always two fewer than a power of two because the two-way decompositions where either S_1 or S_2 is empty are not checked.

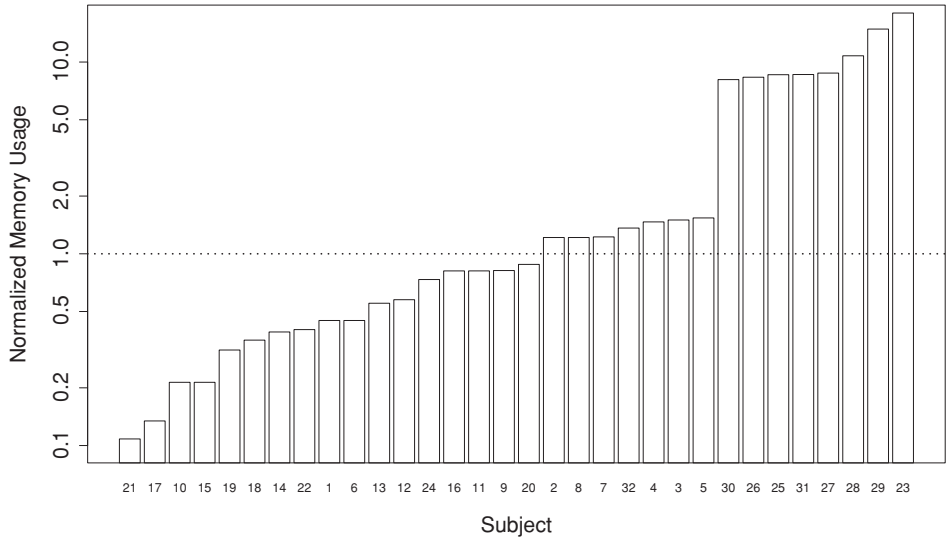

Fig. 1. Memory used by the best decomposition of size 2 for FLAVERS.

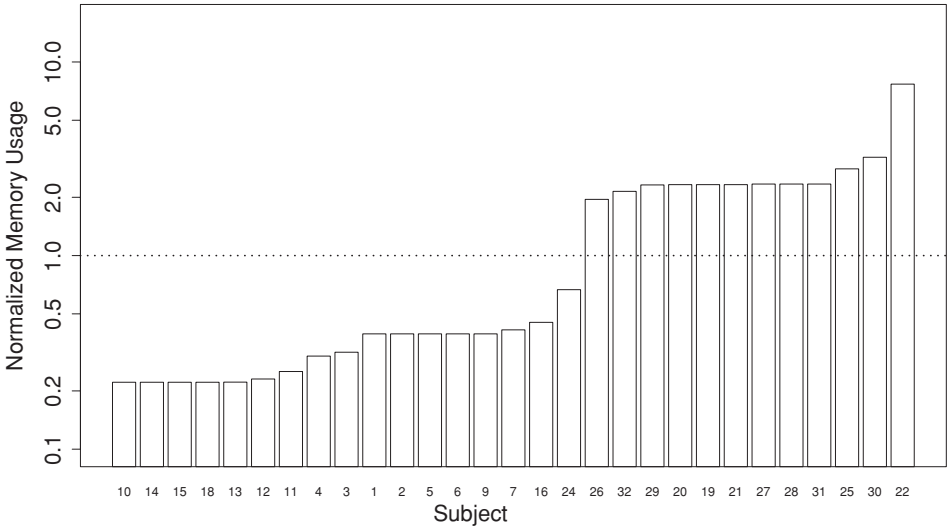

Fig. 2. Memory used by the best decomposition of size 2 for LTSA.

logarithmic. The subject numbers are given on the x-axis so that the reader can compare the performance of the two verifiers for a given subject.

For FLAVERS, the best decomposition is better than monolithic verification on 17 of the 32 subjects. For these 17 subjects, on average the best decomposition uses 48.5% of the memory used by monolithic verification. For the 15 subjects where the best decomposition is worse than monolithic verification, on average the best decomposition uses 654.1% of the memory used by monolithic verification.

Table III. Percentage of Decompositions Worse than Monolithic Verification at Size 2

	FLAVERS		LTSA	
System	Decompositions	Percentage	Decompositions	Percentage
Chiron single	558	96%	496	88%
Chiron multiple	2,286	87%	2,032	88%
Gas Station	120	66%	120	100%
Peterson	6	100%	6	100%
Relay	6	50%	6	83%
Smokers	112	100%	112	100%
Overall	**3,088**	**89%**	**2,772**	**89%**

For LTSA, the best decomposition is better than monolithic verification on 17 of the 30 subjects. For these 17 subjects, on average the best decomposition uses 33.6% of the memory used by monolithic verification. For the 13 subjects where the best decomposition is worse than monolithic verification, on average the best decomposition uses 281.7% of the memory used by monolithic verification.

While there are 17 subjects on which assume-guarantee reasoning is better than monolithic for both FLAVERS and LTSA, these are not exactly the same 17 subjects. There are a total of 12 subjects for which the assume-guarantee approach is better than monolithic verification for both verifiers. Interestingly, the subject on which assume-guarantee reasoning with FLAVERS saves the most memory is a subject on which assume-guarantee reasoning with LTSA does not save memory compared to monolithic verification.

While Figures 1 and 2 show the results for the best decompositions at size 2, it is important to note that the vast majority of decompositions are not better than monolithic verification. Table III shows the percentage of decompositions that are worse than monolithic verification for each system and verifier. For reference, it also gives the total of number of decompositions examined with each verifier for each system.

Overall, with both FLAVERS and LTSA, 89% of the decompositions are worse than monolithic verification. For FLAVERS, there are two systems where these percentages are better than 80%: namely 66% for the Gas Station system and 50% for the Relay system. Even if we only consider those subjects for which there exists a decomposition where assume-guarantee uses less memory than monolithic verification, then these percentages remain basically the same except for the Chiron single system with FLAVERS. On this system the percentage of decompositions that are worse than monolithic verification goes from 96% to 87%. Thus, randomly selecting decompositions would likely not yield a decomposition better than monolithic verification. Furthermore, our intuition on how to select decompositions was not good. Even for the subjects for which there is some decomposition that does save memory, in our preliminary study the decompositions we thought would save memory usually did not. While it might be possible to develop heuristics to aid in finding such a decomposition, when we examined the decompositions that saved memory in our experiments, we did not see any patterns that could be used as the basis for such heuristics.

—For each nonrepeatable task, put the task into S_1 if the task was put into S_1 in the best decomposition at size 2. Otherwise, put the task into S_2.
—For each repeatable task:
  —If the best decomposition for size 2 had both repeatable tasks in S_1, put the repeatable task in S_1.
  —If the best decomposition for size 2 had both repeatable tasks in S_2, put the repeatable task in S_2.
  —If the best decomposition for size 2 had one of the repeatable tasks in S_1 and the other in S_2, look to see if the property treated one of the repeatable tasks in a different way than all the other repeatable tasks.
    —If the property treats all of the repeatable tasks the same, then:
      —If this repeatable task is the repeatable task with the smallest ID, put this repeatable task into S_1 if the repeatable task with the smallest ID was put into S_1 in the best decomposition at size 2. Otherwise, put this repeatable task into S_2.
      —If this repeatable task is not the repeatable task with the smallest ID, put this repeatable task into S_2 if the repeatable task with the smallest ID was put into S_2 in the best decomposition at size 2. Otherwise, put this repeatable task into S_1.
    —If the property treats one of the repeatable tasks different than the other repeatable tasks, then:
      —If this repeatable task is the one that is treated differently, then put this repeatable task into S_1 if its corresponding task in the best decomposition at size 2 was put into S_1. Otherwise, put this task into S_2.
      —If this repeatable task is not the one that is treated differently, then put this repeatable task into S_1 if the repeatable task that is treated differently was in S_2 on the best decomposition at size 2. Otherwise, put this task into S_1.
    —If the property treats two of the repeatable tasks different than other repeatable tasks, then handle the tasks treated in a different way as in the "one" case above and handle the tasks that are not treated in a different way as in the "all" case above.

Fig. 3. Process for generalizing decompositions.

4.2 Does Assume-Guarantee Reasoning Save Memory for Larger System Sizes?

Although assume-guarantee reasoning using learned assumptions saves memory in only about half of the subjects we looked at when the best decomposition is used and finding these best decompositions was expensive, the overall approach was not too onerous. Once the models were built for size-2 systems, on average it required about two minutes to examine one decomposition with FLAVERS and about half a minute to examine one decomposition with LTSA. For larger size systems, however, it would be infeasible to evaluate all two-way decompositions because the number of decompositions to be evaluated increases exponentially and the cost of evaluating each decomposition increases as well. For example, we have several instances where evaluating a single decomposition on a system of size 4 takes over 1 month. Thus, if memory is a concern in verifying a property on a system and if it is important to verify it for a larger size, a reasonable approach might be to examine all decompositions for a small system size and then to generalize the best decomposition for that small system size to a larger system size. In this study, we used just such a generalization approach to evaluate the memory usage of assume-guarantee reasoning for larger system sizes.

Our algorithm for generalizing decompositions from the best decomposition for size 2 is shown in Figure 3. We also considered several other heuristics and discuss them in Section 4.7. At a high level, this algorithm assigns each task into one of two categories, either a task is repeatable (e.g., a customer task in the Gas Station system) or nonrepeatable (e.g., all noncustomer tasks in the Gas Station system). A task, whether repeatable or not, is put in S_1 (S_2) if the best decomposition has the corresponding task in S_1 (S_2). For a larger size system, some tasks are repeated; since these added tasks do not have a corresponding task

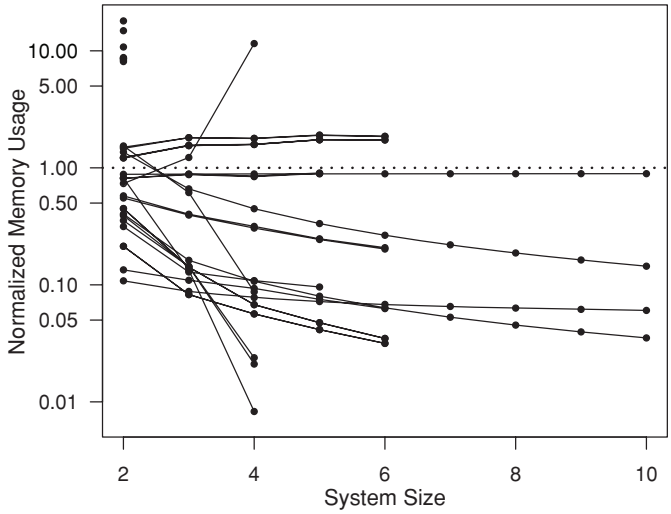

Fig. 4. Memory used by the generalized decompositions for FLAVERS up to size 10.

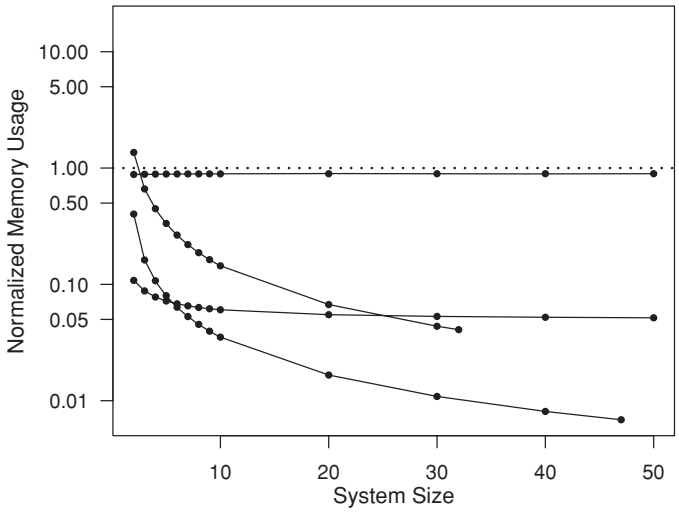

Fig. 5. Memory used by the generalized decompositions for FLAVERS for just those subjects where the largest system size that could be verified is greater than 10.

in the best decomposition at size 2, a heuristic is used to determining whether each should be put in S_1 or S_2. This heuristic is based on the subsystems the repeatable tasks are in for the best decomposition at size 2 and whether or not the property treats the repeatable tasks in a different way from other tasks. For example, using the properties in Table I, subject 8 treats all of the repeatable tasks the same, subject 1 treats one of the repeatable tasks (*artist1*) different than the other repeatable tasks, and subject 28 treats two of the repeatable tasks (*smoker1* and *smoker2*) different than the other repeatable tasks.

Figures 4, 5, and 6 show, as the system size increases, the amount of memory used by assume-guarantee reasoning with generalized decompositions

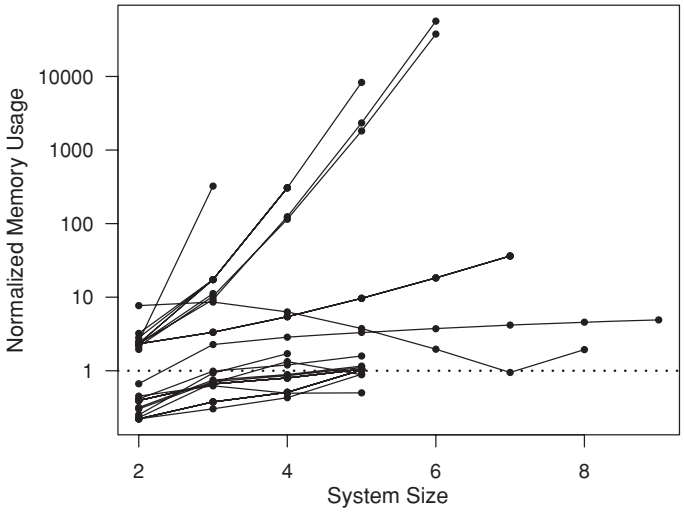

Fig. 6. Memory used by the generalized decompositions for LTSA.

normalized by dividing by the amount of memory used by monolithic verification of the same subject at the same system size. Each solid line represents a single subject. A dotted line at 1.0 has been provided for reference. Figures 4 and 5 show data for FLAVERS. The former shows the data for system sizes less than or equal to 10 while the latter shows the data for only those subjects that could scale above size 10. Note that each line in Figure 5 corresponds to a line that is shown in Figure 4. The two lines in Figure 5 that have data at size 50 correspond to subjects where we actually have data up to size 200. The slopes of the lines for these two subjects do not change significantly above size 50, however. Thus, we have chosen to not show data above size 50 so that details at the smaller sizes can be seen more easily. Figure 6 shows all of the data for LTSA. For 8 subjects with FLAVERS and 3 subjects with LTSA there are single points at size 2. On these subjects, the generalized decompositions run out of memory at size 3. Note that each line stops at the largest system size on which both monolithic verification and assume-guarantee reasoning can verify the corresponding subject. For some subjects either assume-guarantee reasoning or monolithic verification can verify that subject at even larger sizes.

With FLAVERS, if the best decomposition at size 2 is better than monolithic verification, the associated generalized decompositions are usually better than monolithic verification, as seen with 16 of the 17 such subjects. There are 2 subjects on which the best decomposition at size 2 is worse than monolithic verification, but the generalized decomposition is better than monolithic verification at the largest size such a comparison could be made. Thus, with FLAVERS assume-guarantee reasoning using generalized decomposition is better than monolithic verification on 18 of the 32 subjects.

With LTSA, of the 17 subjects on which the best decomposition at size 2 is better than monolithic verification, on only 5 of these is the generalized

decomposition better than monolithic verification at the largest size such a comparison could be made. With LTSA, there are two subjects worth noting. The first is subject 22 where assume-guarantee reasoning is better than monolithic verification at size 7 and worse at all other sizes. On this subject, assume-guarantee reasoning ran out of memory at size 9, but monolithic verification was able to verify the property. The second subject is subject 12, where assume-guarantee reasoning is worse than monolithic verification at size 4 and better at all other sizes up to size 5, the largest size that we were able to try.[3]

If we restrict our attention to just those subjects where the best decomposition at size 2 saved memory, then at larger system sizes FLAVERS used less memory than monolithic verification on 16 of the 17 subjects and LTSA used less memory than monolithic verification on 5 of the 17 subjects. This is a 94% success rate for FLAVERS and a 29% success rate for LTSA, a significant difference.

Figures 4, 5, and 6 illustrate this difference in the performance of the generalized decompositions between FLAVERS and LTSA. With FLAVERS, for the subjects where the best decomposition at size 2 is worse than monolithic verification, assume-guarantee reasoning tends to use increasingly larger amounts of memory, when compared to monolithic, as the system size increases. In other words, the normalized memory usage increases as the system size increases. With FLAVERS, for the subjects where the best decomposition at size 2 is better than monolithic verification, assume-guarantee reasoning tends to save more memory, when compared to monolithic verification, as system size increases. In other words, the normalized memory usage tends to decrease as the system size increase.

This is not true, however, with LTSA. With LTSA, assume-guarantee reasoning tends to use more memory, when compared to monolithic verification, as the system size increases. In other words, the normalized memory tends to increase as the system size increases, regardless of the performance of assume-guarantee reasoning at size 2, although this increase was more pronounced on subjects where the memory used by the best decomposition at size 2 was worse than monolithic verification.

We believe the difference in the performance of assume-guarantee reasoning on the two verifiers is mostly due to the differences in the models used by the two verifiers. For LTSA, each thread is represented by a thread reachability graph. For FLAVERS, each thread is represented by a control flow graph, which is usually smaller and more abstract than a thread reachability graph. FLAVERS adds precision into the model through the use of constraints. Since the number of constraints often does not increase as the system size increases [Dwyer et al. 2004], this means that the model used by FLAVERS often does not increase in size as quickly as the model generated for LTSA. It seems likely that this observed difference resulted from the performance difference between the two verifiers.

[3]Unfortunately, we could not generate the model for this system at size 6. We discuss this issue in Section 4.3.

To summarize the results at larger system sizes, with FLAVERS the best decomposition at size 2 is better than monolithic verification on 17 of the 32 subjects, or about 53% of the time, and the generalized decomposition is better than monolithic verification at the largest size both could verify on 18 of the 32 subjects, or about 56% of the time. With LTSA the best decomposition at size 2 is better than monolithic verification on 17 of the 30 subjects, or about 56% of the time, and the generalized decomposition is better than monolithic verification at the largest size both could verify on 3 of the 30 subjects, or 10% of the time. Thus, the automated assume-guarantee reasoning technique we used was able to save memory on larger size systems for a bit more than half the subjects with FLAVERS and for one tenth of the subjects with LTSA.

4.3 Can Assume-Guarantee Reasoning Verify Properties of Larger Systems than Monolithic Verification Can?

Although using generalized decompositions for assume-guarantee reasoning uses less memory than monolithic verification in some cases, this memory savings might not be sufficient to overcome the state-explosion problem. Thus, for each subject, we tried to determine whether or not assume-guarantee reasoning using generalized decompositions would allow us to verify properties for larger systems than monolithic verification.

Unfortunately, the language processing toolkit[4] [Taylor et al. 1988] used by FLAVERS for generating its models and by INCA for generating the models for LTSA cannot handle the Chiron and Relay systems at larger sizes. Thus, we were unable to determine whether assume-guarantee reasoning using generalized decompositions would allow us to verify properties of larger systems than monolithic verification for some of the subjects associated with these two systems. Thus, we assigned each subject to one of five categories:

(1) Assume-guarantee reasoning can verify a subject at a larger system size than monolithic verification.
(2) It is unknown whether assume-guarantee reasoning can verify a subject at a larger system than monolithic verification because the language processing toolkit could not generate models for a large enough size system for at least one of monolithic verification or assume-guarantee reasoning to run out of memory. We consider it likely that assume-guarantee reasoning can verify larger systems than monolithic verification, however, because assume-guarantee reasoning is better than monolithic verification for the largest system size such a comparison can be made.
(3) Both assume-guarantee reasoning and monolithic verification can verify a subject on systems of size 200, and thus assume-guarantee reasoning would be of little value. In all such cases where this occurred in our experiments, assume-guarantee reasoning is better than monolithic verification at size 200, and thus to be conservative we consider these subjects as likely successes.

[4]This toolkit is old and not easily modifiable.

Table IV. Generalized Decompositions Compared to Monolithic Verification with Respect to Scaling

		FLAVERS		LTSA	
		Number of Subjects	Percentage	Number of Subjects	Percentage
Likely Success	(1) Generalized can scale farther than monolithic	8	25.0%	0	0.0%
	(2) Don't know, but generalized appears better than monolithic	7	21.9%	2	6.7%
	(3) Monolithic scales well, but generalized appears to be better	2	6.3%	0	0.0%
	Subtotal	**17**	**53.1%**	**2**	**6.7%**
Likely Failure	(4) Generalized cannot scale farther than monolithic	13	40.6%	14	46.7%
	(5) Don't know, but generalized appears worse than monolithic	2	6.3%	14	46.7%
	Subtotal	**15**	**46.9%**	**28**	**93.3%**
	Total	**32**	**100.0%**	**30**	**100.0%**

(4) Assume-guarantee reasoning cannot verify a subject at a larger system than monolithic verification.

(5) It is unknown whether assume-guarantee reasoning can verify a subject at a larger system than monolithic verification because the language processing toolkit could not generate models for a large enough size system for at least one of monolithic verification or assume-guarantee reasoning to run out of memory. We consider it unlikely that assume-guarantee reasoning can verify larger systems than monolithic verification, however, because assume-guarantee reasoning is worse than monolithic verification for the largest system size such a comparison can be made.

Table IV shows the number of subjects in each category for FLAVERS and LTSA. We consider the use of generalized decompositions to likely succeed in verifying a property on a larger system than monolithic verification could handle if the subject is in category 1, 2, or 3. Although we consider assume-guarantee reasoning to not be needed if the subject is in category 3, we still count subjects in this category as a likely success. We consider the use of generalized decompositions to likely fail in verifying a property on a larger system than monolithic verification could handle if the subject is in category 4 or 5. As mentioned previously, for LTSA subject 12 is hard to classify since assume-guarantee reasoning is worse than monolithic verification at size 4 and better at all other sizes. Because of language processing issues, we could not build a model for this system at size 6. Since assume-guarantee reasoning is better than monolithic verification at size 5, the largest size such a comparison can be made, we conservatively assigned this subject to category 2.

Although we could demonstrate that assume-guarantee reasoning scales farther than monolithic verification on eight subjects for FLAVERS, six in category 1 and two in category 3, it is also important to look at how much farther assume-guarantee reasoning scales. On five of these eight subjects, assume-guarantee reasoning can verify the subject on a system one size larger, but not two sizes larger, than monolithic verification can. On one of the eight subjects, assume-guarantee reasoning can verify the subject on a system two sizes larger, but not three sizes larger, than monolithic verification can. On the remaining two subjects assume-guarantee reasoning can verify the subject on a system at least three sizes larger than monolithic verification can. On these last two subjects we were able to increase the size of the subject that could be verified from 32 to 35 in one case and from 47 to 50 in the other case. Although there are eight subjects where assume-guarantee reasoning can scale farther than monolithic verification, assume-guarantee reasoning can verify these subjects on systems only slightly larger than the size of the system on which these subjects can be verified monolithically.

Table IV shows a potential success rate of about 53% for FLAVERS and about 7% for LTSA. Note that this rate is the upper bound of the success rate. By looking at just the subjects where we could demonstrate that assume-guarantee reasoning could scale farther, we obtain the lower bound of the success rate: 25% for FLAVERS and 0% for LTSA.

In summary, we expect that assume-guarantee reasoning would be an effective approach for verifying properties at larger system sizes than monolithic verification on at most 53% of the subjects for FLAVERS and on at most 7% of the subjects for LTSA. While a 53% success rate may look encouraging, assume-guarantee reasoning using generalized decompositions did not significantly increase the size of the systems for which we could verify properties. Considering the effort to find the best decomposition at size 2, it is questionable whether or not the benefit of verifying a subject on a slightly larger system size is worth the necessary investment of time.

Although these results are discouraging, we also tried to determine if there was some way to predict the subjects for which assume-guarantee reasoning would likely produce a significant memory savings. Unfortunately, we could not find such a classification, although we do have some observations. One type of constraint used by FLAVERS is a *Task Automaton* (TA). It appears that when the number of TAs needed to prove a property increases as the system size increases, assume-guarantee reasoning based on generalized decompositions tends to use more memory than monolithic verification. Of the 14 subjects where this is the situation, 10 of them are classified as failures in Table IV. Of the 13 subjects where only one TA is needed to prove the property regardless of system size, 4 are classified as failures in Table IV and 3 are in category 3 where assume-guarantee reasoning is not likely to be of much use since monolithic verification scales well. On the remaining 5 subjects, we cannot find a pattern to determine whether assume-guarantee reasoning based on generalized decompositions will perform better than monolithic verification. For LTSA, there are too few successful cases to predict when assume-guarantee reasoning might be successful. Interestingly, there was 1 subject, subject 16,

Table V. System Sizes at Which the Best Decomposition is Known

	FLAVERS		LTSA	
Size	Attempted	Succeeded	Attempted	Succeeded
2	32	32	30	30
3	32	23	30	30
4	24	18	21	21
5	2	2	1	1

where assume-guarantee reasoning for FLAVERS works poorly, but assume-guarantee reasoning for LTSA works well. While our observations may provide some guidance to help determine whether or not assume-guarantee reasoning is likely to increase the size of the system on which a property can be verified, more experimentation is needed before any conclusions can be drawn.

4.4 Are the Generalized Decompositions the Best Decompositions?

These discouraging results were obtained using decompositions that were generalized from the best decomposition on problems of size 2. It is possible that the generalized decompositions we selected are not the best decompositions to use on the larger systems sizes. To investigate this issue, we tried to find the best decomposition for some larger system sizes.

4.4.1 *Comparing the Best-Known Decompositions to the Generalized Decompositions.* In performing these experiments, we encountered a number of two-way decompositions where it took more than a month to learn an assumption.[5] As a result, we imposed an upper bound on the amount of time we spent evaluating a single two-way decomposition to be the maximum of 1 hour and 10 times the amount of time needed to verify that subject monolithically. Thus, for some of the decompositions, assume-guarantee reasoning was not allowed to run until it completed and, as a result, we do not know how good these decompositions are. On every subject where the upper bound on time was reached for some decomposition, we were able to find at least one decomposition for that subject that is better than the generalized decomposition. Table V gives the number of subjects for which we attempted to find the best decomposition at a given system size. It also gives, in the Succeeded column, the number of subjects for which we were able to find the best decomposition at a given system size, meaning those subject for which the time bound was never reached on any decomposition.

Figures 7 and 8 compare, for FLAVERS and LTSA respectively, the memory usage of the generalized decomposition, the best-known decomposition, and monolithic verification at the largest size such a comparison could be made; note that this size varies from subject to subject. In these figures the height of the bars (meaning the top horizontal line on each bar, not the top of the triangles

[5]The time needed to evaluate the decompositions that took more than a month is counted in the 1.54 years of CPU time needed for our experiments. Still, more than 99% of the decompositions required less than 1 day to evaluate. The time used for just the decompositions that took less than 1 day to evaluate added up to over 11 months of CPU time.

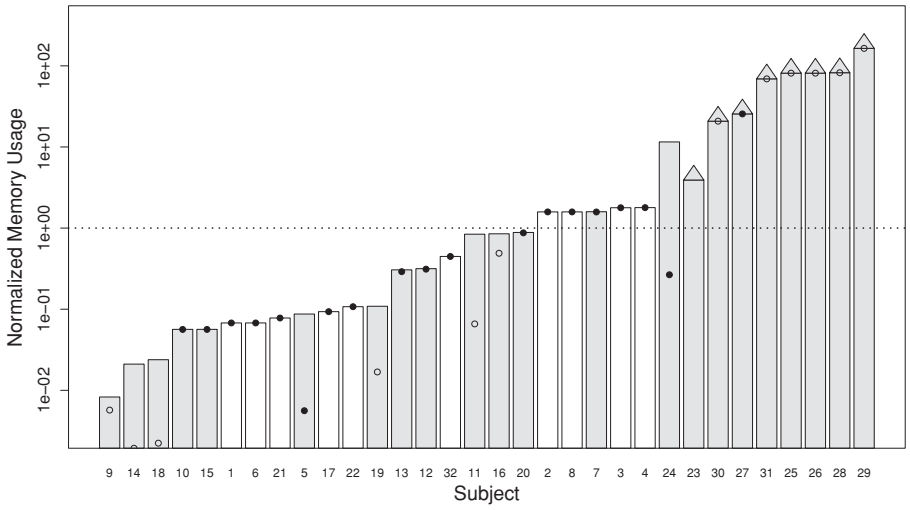

Fig. 7. Memory used by the generalized decomposition compared to the best-known decomposition for FLAVERS.

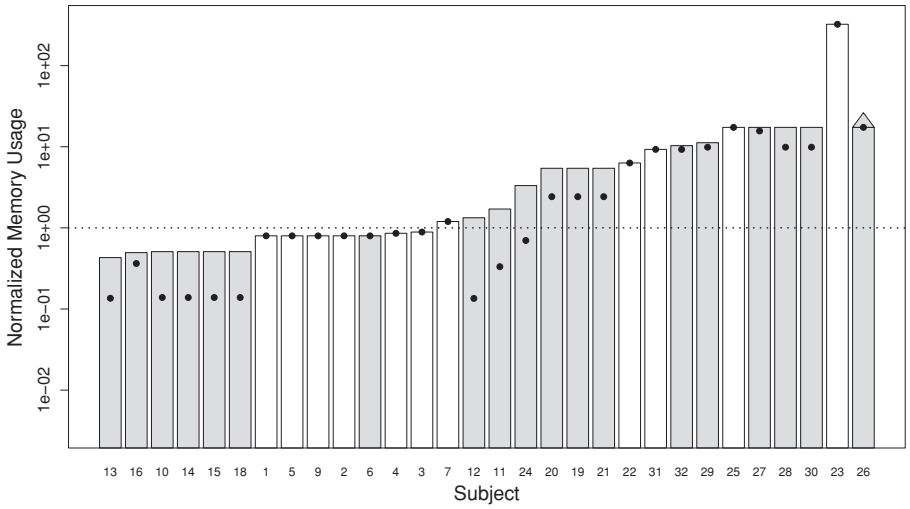

Fig. 8. Memory used by the generalized decomposition compared to the best-known decomposition for LTSA.

for those bars with triangles) shows the amount of memory used by the generalized decomposition, normalized by dividing by the amount of memory used by monolithic verification. The height of the dots shows the amount of memory used by the best-known decomposition, normalized by dividing by the amount of memory used by monolithic verification. A dot that is filled in represents a subject for which we know the best decomposition. A dot that is not filled in represents a subject for which we do not know the best decomposition, meaning there might be better decompositions than our best-known decomposition. Because it is not always clear from the figure whether or not the height of the

bar is the same as the height of the dot, we use a bar with a white background to represent a subject where the generalized decomposition is the same as the best decomposition and a bar with a gray background to represent a subject where the generalized decomposition is not the same as the best decomposition. Bars that are topped with triangles represent subjects where the generalized decompositions ran out of memory on systems with size 3. The height of these bars (not counting the triangles) shows the lower bound on the amount of memory used by the generalized decompositions, meaning the generalized decomposition uses at least as much memory as the height of the bar. Each subject has been labeled with the same number that was assigned to that subject in Table I. For reference, a line at 1.0 has been drawn.

As stated previously, because of the cost involved we did not always obtain data about the best decomposition at the largest system size on which we were able to use the generalized decomposition. For example, for subject 1 with LTSA the generalized decomposition scaled to size 5 but we only found the best decomposition at size 4. So, Figure 8 shows the data at size 4 for subject 1 because this is the largest size for which we have information about the best decomposition, the generalized decomposition, and monolithic verification. For subject 1 with LTSA, the generalized decomposition is better than monolithic verification at size 4 but not at size 5. Thus, Figure 8 shows the generalization decomposition on subject 1 as performing better than monolithic verification, but subject 1 is counted as being in category 5 in Table IV. Because there are multiple subjects for which this is the case, there is not a simple correspondence between the data in Figures 7 and 8.

Note that for LTSA, every subject has a filled dot, meaning we know the best decomposition for each subject at the largest system size we tried to find the best decomposition for that subject. There are still larger system sizes for these subjects for which we did not try to find the best decomposition.

These figures show that using the generalized decompositions is not always optimal with respect to memory usage. With FLAVERS, the generalized decomposition is the best decomposition on 10 of 32 subjects (31%). With LTSA, this is true on 11 of 30 subjects (37%). For some subjects, the difference in memory usage between the generalized decomposition and the best-known decomposition is significant, while in other cases, there is almost no difference. For very few subjects, though, is the generalized decomposition worse than monolithic verification while the best decomposition is better than monolithic. This happened on only one subject with FLAVERS, subject 24, and three subjects with LTSA, subjects 11, 12, and 24.

4.4.2 *Generalizing Decompositions from the Best-Known Decomposition at Larger System Sizes.* For the subjects where the generalized decomposition is not the best decomposition, we were interested in determining if generalizng the best-known decomposition for a system size larger than 2 could be used be used to verify larger systems than can be verified monolithically. Thus, when we found a decomposition for size n ($n > 2$) that was better than the generalized decomposition from size 2, we generalized the best-known decomposition for size n so that it could be used on systems larger than size n. In all such

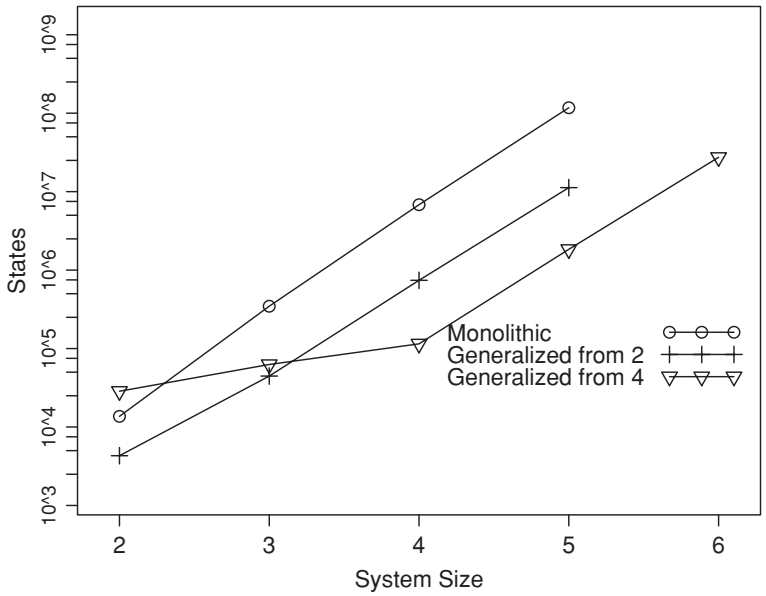

Fig. 9. States explored on subject 19 with FLAVERS.

cases, the generalized decomposition from size n is better than the generalized decomposition from size 2. We also tried taking the best-known decompositions for size n and simplifying them so they could be used on systems of size 2, similar to the process shown in Figure 3, but in reverse. The decompositions for size n, when simplified to size 2, are worse than monolithic verification in all but one case.

In addition we found 3 subjects where there are decompositions that can be used to verify that subject on a larger system size than either monolithic verification or the generalized decompositions from size 2. One of these is subject 19 with FLAVERS. Figure 9 compares the number of states explored using two different generalizations (from size 2 and from size 4) to the number of states explored using monolithic verification. For this subject, the generalized decomposition from size 2 is the best decomposition for size 3, but it is not the best decomposition for size 4. We do not know what the best decomposition is for size 4 because it requires too much time to find. We do know that if we generalize the best-known decomposition for size 4, we can verify this subject on systems with size 6, one size greater than monolithic verification and the generalized decomposition from size 2.

On subject 24 with FLAVERS, the generalized decompositions from size 2 are worse than monolithic verification. We were able to find a decomposition that allowed us to verify this subject on the system with size 7, one size larger than monolithic verification. This decomposition, however, was not easy to find. When looking at the best decompositions for size 2, 3, 4, and 5, we noticed that there is a pattern to the best decompositions that depends on whether or not the size of the system is odd or even. For this subject, Figure 10 compares the

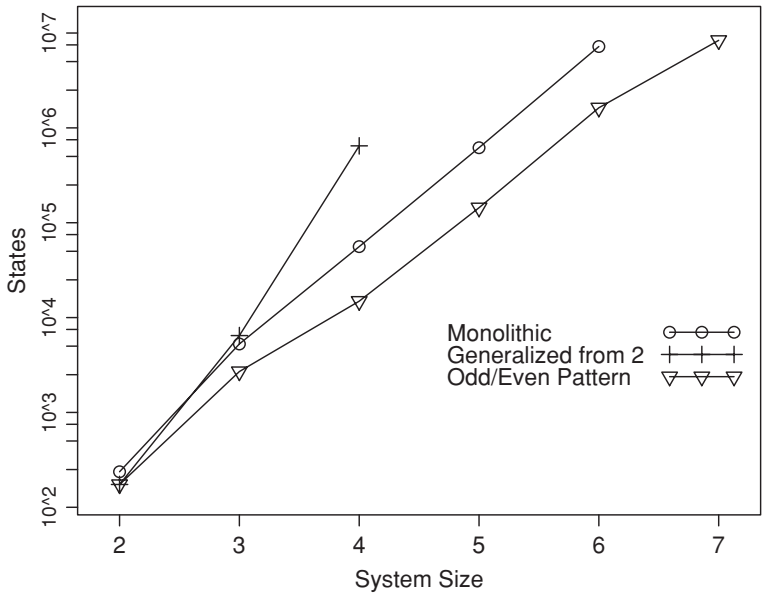

Fig. 10. States explored on subject 24 with FLAVERS.

number of states using the generalized decomposition from size 2, the decompositions based on the odd/even pattern, and the monolithic verification.

For the third subject, subject 11 with FLAVERS, monolithic verification and assume-guarantee reasoning using the generalized decomposition from size 2 can verify this subject on a system of size 5, but not 6. Using the generalized decomposition from size 3 allows assume-guarantee reasoning to verify this subject on a system of size 6.

To summarize, although finding the best decomposition at a small system size and generalizing it so that it is applicable for larger system sizes was not too costly a process, it often does not produce the best decomposition at those larger sizes. There are subjects where using decompositions other than the generalized decomposition from size 2 allowed us to verify properties on larger systems than when we used those generalized decompositions. Because we were unable to find heuristics that enabled us to find these better decompositions, we tried all two-way decompositions for larger system sizes, a process that is probably too costly to be useful in practice.

4.4.3 *Discussion.* While we tried to find the best decomposition for all subjects at some larger systems sizes, because of the costs involved we did not always attempt this at the largest size for each subject in our study. At the largest size we could compare the generalized and the best decomposition, they were the same on 10 of 32 subjects with FLAVERS and on 11 of 30 subjects with LTSA. With FLAVERS, for 15 of the 32 subjects we were not able to find the best decomposition at the largest system size we attempted but we were able to find a decomposition that uses less memory than the generalized decompositions

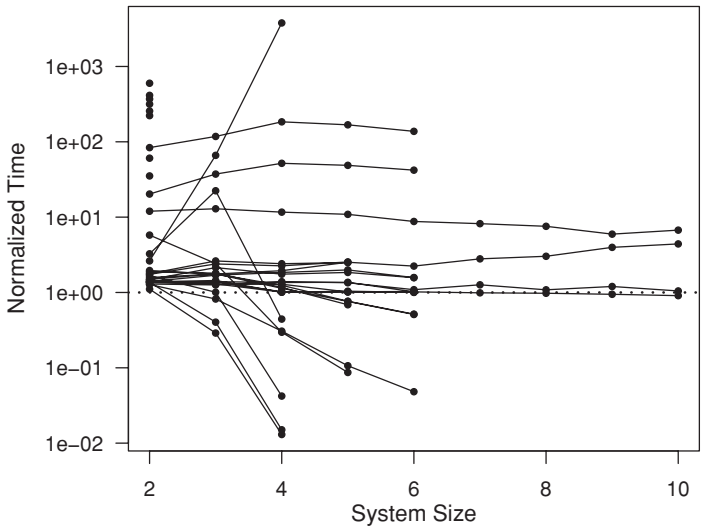

Fig. 11. Time used by the generalized decompositions for FLAVERS up to size 10.

from size 2. With LTSA, for all of the subjects we were able to find the best decomposition at the largest system size we attempted.

As stated previously, our use of generalized decompositions from size 2 often did not allow us to verify properties on larger systems than monolithic verification and, when it did, it only allowed us to verify those properties on systems 1 or 2 sizes larger. Had we been able to find the best decomposition for every subject at every system size, we do not think that our results would be significantly different. There might be subjects, like the three discussed in Section 4.4.2, where decompositions other than the ones we examined would have allowed us to verify those subjects at larger system sizes than monolithic verification. But, since there are few subjects on which the best-known decomposition is significantly better than the generalized decomposition, as shown in Figures 7 and 8, we doubt that there exist other decompositions that would have allowed us to verify properties on significantly larger systems than monolithic verification.

4.5 Does Assume-Guarantee Reasoning Save Time?

Although assume-guarantee reasoning was not usually successful in increasing the size of the systems on which properties could be verified, if assume-guarantee reasoning could reduce the time needed for verification, it might still be worth using. When we compared the time used by assume-guarantee reasoning to the time used by monolithic verification, including the time to build the artifacts (e.g., TFGs) but not counting the time used by the language processing toolkit, we found that assume-guarantee reasoning with FLAVERS uses less time on 11 of the 32 subjects (34.4%) and that assume-guarantee reasoning with LTSA uses less time on 7 of the 30 subjects (23.3%)

Figures 11, 12, and 13 show the amount of time used by assume-guarantee reasoning with generalized decompositions from size 2 normalized by dividing

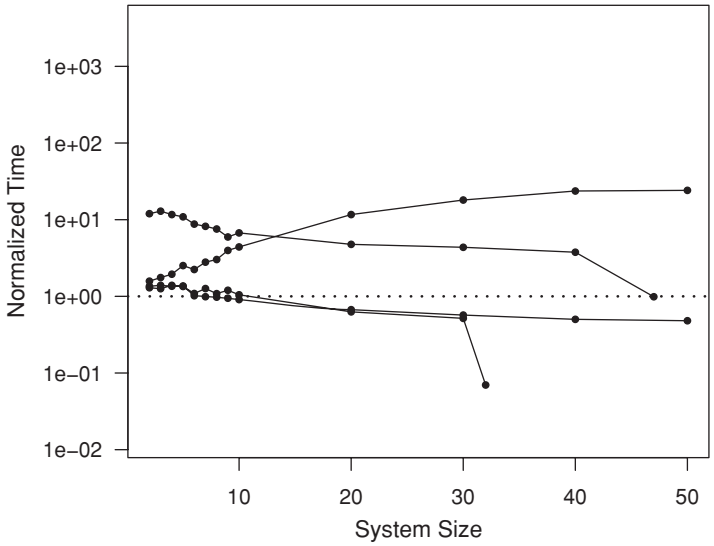

Fig. 12. Time used by the generalized decompositions for FLAVERS for just those subjects where the largest system size that could be verified is greater than 10.

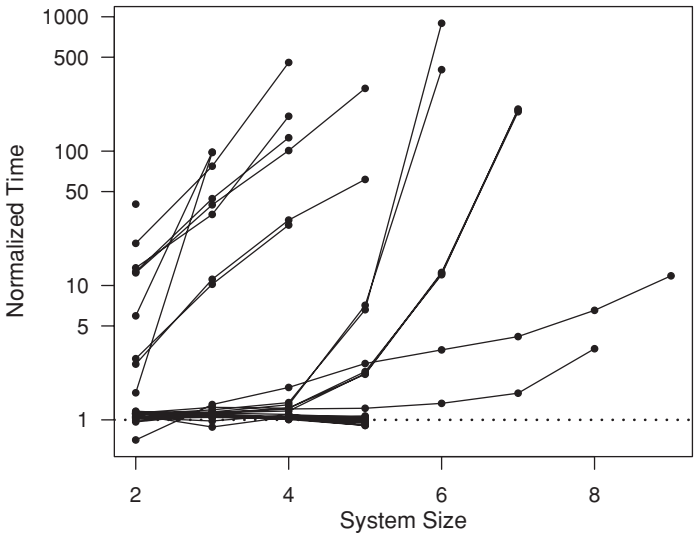

Fig. 13. Time used by the generalized decompositions for LTSA.

by the amount of time used by monolithic verification as the size of the systems is increased. Each solid line represents a single subject. A dotted line at 1.0 has been provided for reference. Figures 11 and 12 show data for FLAVERS. The former shows the data for system sizes less than or equal to 10 while the latter shows the data for only those subjects that could scale above size 10. Note that each line in Figure 12 corresponds to a line that is shown in Figure 11. The two lines in Figure 12 that have data at size 50 correspond to subjects where

we actually have data up to size 200. As before, the slopes of the lines for these two subjects do not significantly change above size 50, and thus we have chosen not to show data above size 50 so that details at the smaller sizes can be more easily seen.

Figure 13 shows all of the data for LTSA. As before, for 8 subjects with FLAVERS and 3 subjects with LTSA there are single points at size 2. On these subjects, the generalized decompositions from size 2 runs out of memory at size 3.

There are two subjects worth noting in Figure 12, the two subjects where the slope suddenly changes before the last data point. These are subjects 22 and 33; they correspond to systems where monolithic verification runs out of memory at the next largest system size and needs almost all of the available memory to verify the subject at the largest size at which it can be verified. It seems likely that the Java garbage collector needs to run more often and, thus, introduces extra overhead to monolithic verification, slowing it down significantly. This might explain the drastic change in slope seen for these two subjects.

To summarize, on the subjects for which assume-guarantee reasoning uses less time than monolithic verification, it sometimes uses significantly less time. When assume-guarantee reasoning uses more time than monolithic verification, it often uses significantly more time, particularly for LTSA. A large portion of this time cost is due to the learning algorithm we used, and this is discussed in the next section.

4.6 What Is the Cost of Using the L* Algorithm?

We also tried to investigate whether or not using the L* algorithm to learn assumptions increased the cost of assume-guarantee reasoning. To do this, we first determined, for each subject, the cost of assume-guarantee reasoning when the L* algorithm is used to learn an assumption. At the end of each verification done with assume-guarantee reasoning, we saved the assumption that was used to complete the assume-guarantee proof. These assumptions were then used to evaluate the cost of assume-guarantee reasoning when assumptions are not learned. For each subject with property P, this cost was determined by checking $\langle A \rangle\ S_1\ \langle P \rangle$ and $\langle true \rangle\ S_2\ \langle A \rangle$, letting A be the assumption that was previously learned when P was verified using assume-guarantee reasoning with the L* algorithm. For each subject, we compared the time and memory costs for the largest sized system on which that subject could be verified using automated assume-guarantee reasoning with the decompositions generalized from size 2.

4.6.1 *Memory Cost of Using the L* Algorithm.* For a subject with property P, we consider the amount of memory used during assume-guarantee reasoning with the L* algorithm to be the maximum number of states explored when the teacher answers a query or conjecture of the L* algorithm. We consider the amount of memory used during assume-guarantee reasoning without the L* algorithm to be the maximum number of states explored when verifying $\langle A \rangle\ S_1\ \langle P \rangle$ and $\langle true \rangle\ S_2\ \langle A \rangle$, where A is the assumption that was previously learned by the L* algorithm.

For FLAVERS, the amount of memory used by the two approaches is the same on 29 of the 32 subjects. On the remaining three subjects, assume-guarantee reasoning without the L* algorithm uses 78.6%, 86.6%, and 96.7% of the memory used by assume-guarantee reasoning with the L* algorithm. On two of these three subjects, the amount of memory used by assume-guarantee reasoning with and without the L* algorithm is greater than the amount of memory used by monolithic verification. On the third subject, assume-guarantee reasoning without the L* algorithm uses 0.54% of the memory of monolithic verification compared to 0.69%, with the L* algorithm. Since this difference is so small on this third subject, we doubt that learning an assumption affected the results of whether or not assume-guarantee could verify properties of larger system sizes than monolithic verification. Overall, it does not appear that our use of learning significantly impacted the results of our experiments with FLAVERS.

For LTSA, the amount of memory used by the two approaches is the same on 29 of the 30 subjects. On the remaining subject, assume-guarantee reasoning without the L* algorithm uses 14.5% of the memory used by assume-guarantee reasoning with the L* algorithm. On this subject, assume-guarantee reasoning with the L* algorithm uses 491.1% of memory that monolithic verification uses, but without the L* algorithm uses only 71.4% of memory that monolithic verification uses. Thus, on this subject, our use of learning could have affected whether or not assume-guarantee could verify a larger system than monolithic verification. When we looked at the assumption generated for this system at size 2 and size 3, we could not see a way to generalize the assumption to make it applicable on larger system sizes. As a result, to verify this subject using assume-guarantee reasoning, it would probably be necessary to use the L* algorithm or some other automated approach.

To summarize, for very few subjects does assume-guarantee reasoning with learning use more memory than verifying that subject with assume-guarantee reasoning using a supplied assumption. Even for the small number of subjects where there is memory overhead as a result of our use of the L* algorithm to learn an assumption, we doubt that this overhead had a significant impact on the results of whether or not assume-guarantee reasoning could verify properties of larger systems than monolithic verification.

4.6.2 *Time Cost of Using the L* Algorithm.* For a subject with property P, we consider the amount of time used during assume-guarantee reasoning with the L* algorithm to be the amount of time needed to verify that subject, including the time to build the artifacts (i.e, the TFGs, LTSs, etc.) and run the L* algorithm. We consider the amount of time used during assume-guarantee reasoning without the L* algorithm to be the amount of time needed to verify $\langle A \rangle\ S_1\ \langle P \rangle$ and $\langle true \rangle\ S_2\ \langle A \rangle$, including the time needed to build the artifacts, where A is the assumption that was previous learned by the L* algorithm. In neither case do we count the time used by the language processing toolkit.

Figures 14 and 15 show the percentage of time that is spent learning an assumption compared to the size of the assumption that was learned. When the size of the learned assumption is small, fewer than 10 states, less than 50% of the total verification time is spent learning the assumptions. When the size

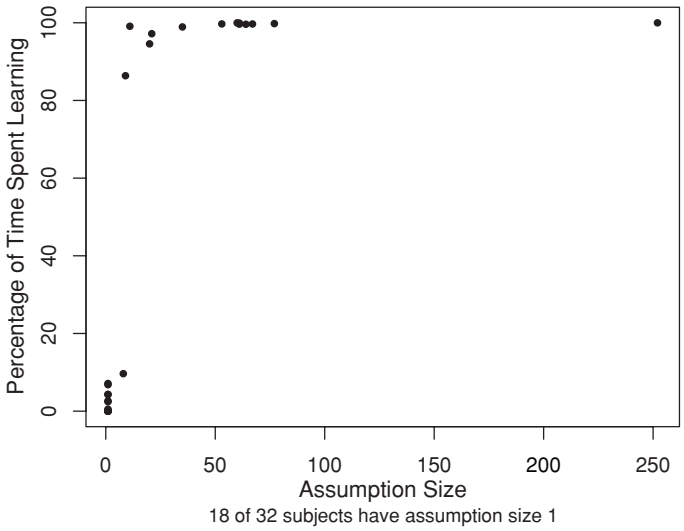

Fig. 14. Percentage of time spent learning for FLAVERS.

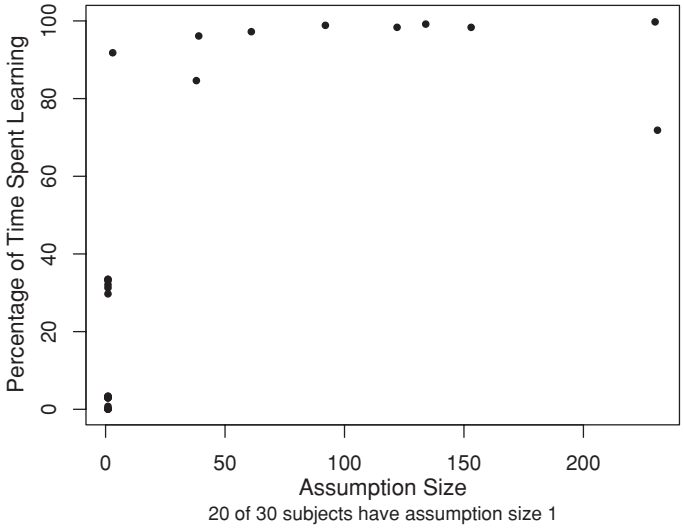

Fig. 15. Percentage of time spent learning for LTSA.

of the learned assumption is larger than 10 states, however, usually over 90% of the verification time is spent learning the assumptions, a substantial overhead. Thus, our use of learned assumptions had a significant impact on the time cost of assume-guarantee reasoning for many subjects.

4.6.3 *Reducing the Cost of Using the L* Algorithm.* Because of this time overhead, we looked at two ways to reduce the cost of automatically learning an assumption. First, since we were able to use generalized decompositions to apply assume-guarantee reasoning to larger-sized systems, we tried generalizing

the assumptions in a similar fashion. When the learned assumption was small, the learning algorithm did not add significant overhead to verification, so reducing this cost would not have had a significant impact in reducing the cost of verification. When the learned assumption was large, however, it was difficult to understand what behavior the assumption is trying to capture. Without such an understanding, it is not possible to determine what the assumption should be for a larger-sized system. As a result, we were unable to use generalized assumptions to reduce the cost of automated assume-guarantee reasoning.

Second, we tried to apply the work of Groce et al. [2002] to help reduce the cost of using the L* algorithm. This work presented a technique for initializing some of the L* algorithm's data structures for Angluin's version of the L* algorithm when given an automaton that recognizes a language close to the one being learned. In our experiments, initializing these data structures in Angluin's version of the L* algorithm did not offer any performance benefits over using Rivest and Schapire's version of the L* algorithm, which has better worst-case bounds. We have been unable to find a similar technique to initialize the data structures of Rivest and Schapire's version of the L* algorithm because of the constraints this version places on its data structures.

4.6.4 *Summary.* Using the L* algorithm to learn an assumption often increases the time needed but rarely increases the memory needed to complete an assume-guarantee proof compared to the cost of completing a proof using a supplied assumption. The overhead for using an automated assumption generation technique, however, is probably unavoidable on several of our subjects. Some of the learned assumptions are very large: in fact, one has over 250 states. For such systems, we suspect that small assumptions do not exist that can be used to complete the assume-guarantee proof and analysts cannot be expected to develop these assumptions manually. Thus, some automated support is needed to make assume-guarantee reasoning practical on these systems.

4.7 Threats to Validity

Although our experiments examined several systems in detail, they are still limited in scope: we used two finite-state verifiers, one assume-guarantee reasoning technique, and a small number of systems. Even in this limited context, our experiments were expensive to perform.

Although we used only two verifiers, we expect that using the L* algorithm to learn assumptions with other verifiers will produce similar results. This conjecture is consistent with the results of Alur et al. [2005] for NuSMV [Cimatti et al. 2002] in which they found some subjects where assume-guarantee reasoning verifies a larger system than monolithic verification and other subjects where assume-guarantee reasoning uses more memory than monolithic verification.

We looked at only one assumption generation technique, which influenced the assumptions used in completing the assume-guarantee proofs. While other assumptions could be used to complete assume-guarantee proofs in our examples, automated support to help find assumptions is necessary to make assume-guarantee reasoning useful in practice. Additionally, we expect that other assumption generation techniques based on two-way decompositions

[e.g., Giannakopoulou et al. 2002; Barringer et al. 2003; Alur et al. 2005; Chaki et al. 2005] would produce assumptions similar to the ones generated by the algorithm we used. Since discharging the premises of the assume-guarantee rules tended to be the most expensive part of the analysis with respect to memory, we suspect that using these other techniques will not produce better results. Although automated techniques based on assume-guarantee rules that allow for more than two-way decompositions [e.g., Inverardi et al. 2000; Henzinger et al. 2003; de la Riva and Tuya 2004] might perform better with respect to memory, there has not yet been enough empirical evaluation of these techniques to draw any conclusions.

Additionally, we looked at only a small number of systems that are mostly based on a client-server architecture and where scaling is achieved by replicating the number of clients. This allowed us to easily increase the size of the system to investigate the effects of scaling on assume-guarantee reasoning. Primarily looking at one kind of architecture, however, might limit the generality of our results. Alur et al. [2005], however, looked at systems with different architectures and had results that were similar to ours.

We also used one generalization approach, shown in Figure 3, when looking at larger system sizes, and that could affect our results. In particular, in the case where none of the repeatable tasks are treated in a different way in the property, we put one task into S_1 and the rest into S_2 (or vice versa). For this case, we also looked at splitting the tasks evenly between S_1 and S_2, either by putting the first half of the repeatable tasks in S_1 and the rest in S_2 or by putting the repeatable tasks with odd IDs in S_1 and the rest in S_2 (or vice versa, in both cases). For most of the subjects, these alternative generalization approaches made little difference in the amount of memory used. For one subject with FLAVERS, these alternate generalizations used between three and four times the memory as the one generated using the algorithm shown in Figure 3. For another subject with FLAVERS, the alternative generalizations used between one third and one half the memory as the one generated using the algorithm shown in Figure 3. Except for these few outliers, the alternative generalization approaches we tried did not produce significantly different results, and thus we do not expect other generalization approaches to be more successful.

Despite these threats, our experiments are a careful study of one automated assume-guarantee reasoning technique and raise doubts about the usefulness of assume-guarantee reasoning as an effective compositional analysis technique.

5. RELATED WORK

Although assume-guarantee reasoning has been proposed as a possible solution to the state-explosion problem [Jones 1983; Pnueli 1984], and many assume-guarantee frameworks have been developed [e.g., Shurek and Grumberg 1990; Grumberg and Long 1994; Abadi and Lamport 1995], the difficulty in generating assumptions to complete assume-guarantee proofs has made evaluating assume-guarantee reasoning difficult. Several case studies have been performed [e.g., Henzinger et al. 1998; McMillan 1998; Fournet et al. 2004],

but these have been limited to small systems because the assumptions were developed manually. Still, both Henzinger et al. and McMillan showed how assume-guarantee reasoning can be used to verify larger systems than monolithic verification using SMV [McMillan 1993] and Mocha [Alur et al. 1998], respectively. Fournet et al. implemented their algorithm for Zing [Andrews et al. 2004] but did not compare their approach to monolithic verification, however, so it is unknown if it offers a benefit over monolithic verification.

Recent work on automated assumption generation has made case studies easier to perform. Several approaches, like ours, are based on the work of Cobleigh et al. [2003], which uses the L* algorithm to learn assumptions. The work of Barringer et al. [2003] extends this approach to use symmetric assume-guarantee rules. Chaki et al. [2004] implemented this approach, and their experimental results showed that, although using the symmetric rule reduced the memory needed compared to the non-symmetric rule presented in Cobleigh et al. [2003], using the symmetric rule could increase the time needed.

Chaki et al. [2005] developed an algorithm based on the L* algorithm for learning tree automata for checking simulation conformance. They examined eight properties of one system and their automated approach always used less time and memory than monolithic verification. They looked at a protocol system with only two components and, thus, never had to consider the decomposition problem, as we did. Since their evaluation of the one system consistently showed improved performance, which is inconsistent with our results, it would be interesting to see a more extensive evaluation of this approach.

Alur et al. [2005] adapted the L* algorithm for use with NuSMV. They found that some of the properties could be verified using assume-guarantee reasoning but not verified monolithically. Some of these properties are for scalable systems and, on these systems, they were able to increase the size of the system that could be verified by one or two. They did not, however, determine if assume-guarantee reasoning can scale farther than this, but, based on their data, it seems unlikely. Alur et al. also reported on one property where assume-guarantee reasoning used more time and memory than monolithic verification.

The work of Giannakopoulou et al. [2002] also computes assumptions, but requires exploring the entire state space of S_1. The scalability of this approach has been compared to our approach only on one small example, and our approach used less memory. Because our approach does not require exploring the entire state space of S_1, we expect that it will use less memory in practice.

Gheorghiu et al. [2007] extended the approach we used in this article so that the alphabet of the assumption is refined during the learning process. Instead of starting with the largest possible alphabet, this approach starts with a smaller alphabet and adds events to it as needed until the property being verified is either shown to hold or to not hold. With 2-way decompositions, in the cases they examined, this approach sometimes used less time and sometimes used less memory then our approach. But, in all the cases they considered, this approach was worse than monolithic verification. Gheorghui et al. also extended their approach to use n-way decompositions, where n is the number of tasks in the system. With n-way decompositions, this approach used less

memory than monolithic verification in about half of their examples and used more time than monolithic verification in almost all of their examples. This approach shows promise because assume-guarantee reasoning with n-way decompositions tended to perform better, in terms of both time and memory, when compared to monolithic verification, as system size increased.

Chaki and Strichman [2007] showed how the L* algorithm can be optimized in the context of assume-guarantee reasoning to reduce the number of queries needed. This work presents three optimizations, including one that minimizes the assumption alphabet, similar to the algorithm presented in Gheorghiu et al. [2007]. Chaki and Strichman evaluated these optimizations by verifying ten properties of one system and found that they reduced the time needed for verification when compared to using the L* algorithm without the optimizations. The system they looked at was a protocol and only had two components and, as a result, they did not consider the decomposition problem, as we did. Additionally, they did not compare their approach to monolithic verification.

Other approaches have been proposed based on assume-guarantee proof rules that allow a system to be decomposed into an arbitrary number of subsystems. Inverardi et al. [2000] showed how to derive assumptions automatically for systems specified in CHAM [Berry and Boudol 1992] to check for freedom from deadlock using assume-guarantee reasoning. This approach has a better worst-case memory bound than monolithic verification but it does not improve upon the worst-case time bound for monolithic verification. This work, however, does not provide an empirical evaluation of the approach, so it is unknown if it offers a benefit over monolithic verification in practice.

Flanagan and Qadeer [2003] developed an automated assume-guarantee reasoning approach for CALVIN [Flanagan et al. 2005] that works on systems with an arbitrary number of tasks that communicate using a shared-memory model. They showed how the worst-case performance of their approach is an improvement over the worst-case performance of monolithic verification, but did not perform empirical studies. Henzinger et al. [2003] built upon this approach and added counterexample guided abstraction refinement [Clarke et al. 2000] for use with BLAST [Henzinger et al. 2002]. Thread-modular reasoning in BLAST is incomplete, meaning there are some properties that can be verified by monolithic analysis but not by the compositional approach. Still, they used this approach to successfully prove properties of several systems. They did not report on how their approach compared to monolithic verification, however.

Data mining techniques [Agrawal et al. 1993] have been used by de la Riva et al. [2001] for building assumptions for SA/RT models (which resemble StateCharts [Harel et al. 1990]). This work was later extended to allow assumptions for multiple components to be generated simultaneously using the assumptions that have already been generated to prune the search space [de la Riva and Tuya 2004]. This approach was applied to one system, but not compared to monolithic verification.

Jeffords and Heitmeyer [2003] used an invariant generation tool to generate invariants for components that can be used to complete an assume-guarantee proof. Although 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. Still, in their experiments, their approach provided a time savings over monolithic verification. They did not, however, report on memory usage.

Another compositional analysis approach that has been advocated is Compositional Reachability Analysis (CRA) [e.g., Yeh and Young 1991; Giannakopoulou et al. 1999]. CRA incrementally computes and abstracts the behavior of composite components using the architecture of the system as a guide to the order in which to perform the composition. CRA can be automated and in some case studies [e.g., Cheung and Kramer 1996] has been shown to reduce the cost of verification. Constraints, both manually supplied and automatically derived, can help reduce the cost of CRA [Cheung and Kramer 1996], but determining how to apply CRA to effectively reduce the cost of verification still remains a difficult problem.

6. CONCLUSIONS

Assume-guarantee reasoning has been proposed as an approach to address the state-explosion problem. There are two major obstacles that are encountered when trying to apply assume-guarantee reasoning to a system. First, if the assume-guarantee rule cannot handle an arbitrary number of subsystems, then a decomposition of the system must be selected in which its subsystems are divided into a suitable number of pieces based on the assume-guarantee rule being used. Second, once a decomposition is selected, it may be difficult to manually find assumptions to complete the assume-guarantee proof. Recent work in assume-guarantee reasoning allows assumptions to be generated automatically, thus removing one of these obstacles to its use. In the study described here we examined *all* two-way decompositions to find the best one with respect to memory use and then generalized those best decompositions to make them applicable on larger system sizes. We evaluated whether or not this approach saved memory over monolithic verification, and, if so, whether or not it allowed larger size systems to be verified. We also evaluated our generalization approach and the cost of automatically generating assumptions in the verification process.

Unfortunately, the results of our experiments are not very encouraging. It is perhaps not surprising that, for the vast majority of decompositions, more states were explored when we used assume-guarantee reasoning than when we used monolithic verification. More significantly, for the subjects at the smallest size, in about half the cases we examined there were no decompositions for which fewer states are explored by the assume-guarantee reasoning approach we used than by monolithic verification. Thus, it is not clear how analysts will be able to identify the cases where assume-guarantee reasoning might be beneficial or how they will be able to find the appropriate decompositions in those cases.

For the subjects where assume-guarantee reasoning could save memory, we were interested in determining if this memory savings would be substantial enough to allow us to verify properties on larger systems than can be verified monolithically. Since it is impractical to examine all two-way decompositions for larger system sizes, we used a generalization approach. For each

property, we found the best decomposition for a small system size and then generalized that best decomposition so it could be used on larger system sizes. Using this approach, we found that even when assume-guarantee reasoning can save memory over monolithic verification, it rarely saved enough memory to allow properties to be verified on larger systems. The memory savings were sufficient to allow verification of larger systems on only eight of the 32 subjects for FLAVERS and on none of the 30 subjects for LTSA. Furthermore, for the cases where assume-guarantee reasoning did verify properties at a larger system than monolithic verification, it did not significantly increase the size of the systems for which properties could be verified. Although there may be systems for which it is useful to verify a slightly larger system size, the overhead of applying this assume-guarantee reasoning technique makes it questionable whether this technique can be effectively used in practice.

Of course, there are decompositions other than the generalized ones that could have been tried on larger systems. In fact, we examined several such decompositions and found some examples where these alternative decompositions could be used to verify larger systems than we were able to verify using the generalized decompositions. Unfortunately, we were unable to find such decompositions intuitively and we did not observe any pattern that could be used to select a good decomposition for a given system.

When we initiated this study, we did not expect that assume-guarantee reasoning would save memory in all cases. We were surprised, however, to discover that in about half of our subjects, assume-guarantee used more memory than monolithic verification, no matter what decomposition was selected. In many cases, this additional cost was due to the assumption learned. While the assumptions were almost always smaller than the subsystems they replaced (i.e., $|A| < |S_2|$), they often allowed behavior that did not occur in S_2. Thus, checking $\langle A \rangle\ S_1\ \langle P \rangle$ was more expensive than checking $\langle \mathit{true} \rangle\ S_1 \parallel S_2\ \langle P \rangle$. (Details on the size of the assumptions and S_2 can be found in the Electronic Appendix for this article.)

Although these results are preliminary, they raise doubts about the usefulness of assume-guarantee reasoning as an effective compositional analysis technique. Although automated assume-guarantee reasoning techniques can make compositional analysis easier to use, determining how to apply these techniques most effectively is still difficult, sometimes expensive, and not guaranteed to significantly increase the sizes of the systems that can be verified.

These results, although discouraging, indicate several directions for future work. The learning algorithm we used converges on the weakest possible assumption [Giannakopoulou et al. 2002], that is, the assumption that allows the most behavior. As stated previously, the high cost for assume-guarantee reasoning was largely due to the fact that the learned assumptions often allowed behaviors that did not occur in the subsystems that they replaced. Thus, one possible direction for future work is to develop approaches that try to learn more specific assumptions than the weakest possible assumption. Recent work on minimizing the alphabets of the learned assumptions seems like a promising step in this direction [Chaki and Strichman 2007; Gheorghiu et al. 2007].

The assume-guarantee rule we used requires that a system under analysis be divided into two subsystems. Rules that allow decomposition into an arbitrary number of subsystems might perform better [Chaki et al. 2004]. Thus, another possible direction for future work is to develop and evaluate such rules.

A third direction for future work is to develop heuristics to help analysts determine when assume-guarantee reasoning is likely to save memory over monolithic verification. In the study reported here, all the systems that we analyzed use a client-server architecture. Systems with other architectures, however, might be more amenable to assume-guarantee reasoning. For instance, a research group at NASA successfully demonstrated that assume-guarantee reasoning could be applied to a system in which a visiting vehicle does autonomous rendezvous and docks with a space station [Brat et al. 2006]. This system was built from two large subsystems that communicated with each other via a small interface. In this case study, assume-guarantee reasoning was able to verify properties that could not be verified monolithically. More experimentation with different architectural models is needed to determine if assume-guarantee reasoning is more effective when applied to certain architectures.

Although assume-guarantee reasoning has been advocated for over twenty years as a way to lessen the effects of the state-explosion problem and recent work in automated assumption generation has made assume-guarantee reasoning easier to apply, our work shows that assume-guarantee reasoning often does not result in a time or memory savings and it is difficult to find a decomposition that is effective or even to know when one exists. These results provide insight into research directions that should be pursued and highlight the importance of further experimental evaluation of compositional analysis techniques.

APPENDIX

In this section, we describe how the teacher for the L* algorithm can be implemented using FLAVERS. To do this, we first need to provide a more detailed description of FLAVERS. Full details of FLAVERS are given in Dwyer et al. [2004].

A.1 More Details about FLAVERS

Consider the example shown in Figure 16, which shows an elevator system in Ada. The system has two tasks, a car and a controller, and a variable x that is shared by both tasks. In this system, the variable x is first set by both tasks and then the tasks rendezvous on sync. This ensures that x is set, but because of the race condition, x could be either true or false when the rendezvous sync occurs. If x is true when sync occurs, then, through the rendezvous open_doors and close_doors, the controller instructs the car to first open and then close its doors. Once that is done, the rendezvous move_car occurs, which causes the car to move. If x is false when sync occurs, then only the rendezvous move_car occurs. Note that this system assumes that the car's doors are closed at the beginning.

```
task body car
  x := true;
  accept sync;

  if (x) then
    accept open_doors;
  end if;

  if (x) then
    accept close_doors;
  end if;

  accept move_car;
end car;
```

```
task body controller
  x := false;
  car.sync;

  if (x) then
    car.open_doors;
  end if;

  if (x) then
    car.close_doors;
  end if;

  car.move_car;
end controller;
```

Fig. 16. Elevator system in Ada.

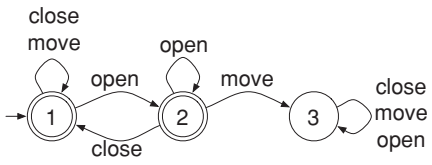

Fig. 17. Example property for FLAVERS.

The properties that FLAVERS verifies need to be expressed as FSAs that represent sequences of events that should (or should not) happen on any execution of the system. One property that should hold on the elevator system is that the car should never move while its doors are open. The FSA for this property is shown in Figure 17, in which the events close, move, and open refer to the rendezvous close_doors, move_car, and open_doors, respectively. State 1 represents the state where the car's doors are closed, state 2 represents the state where the car's doors are open. The transition on move from state 2 to state 3 represents the car moving when its doors are open. State 3, the only nonaccepting state, represents a violation of the property, since the only way to enter it is by having the elevator move while the car's doors are open.

To verify a property, FLAVERS uses a model of the system based on annotated CFGs. Annotations are placed on nodes of the CFGs to represent events that occur during execution of the actions associated with a node. The CFGs for the car and controller tasks are shown in Figures 18 and 19, respectively.

Since the efficiency of FLAVERS' verification is dependent on the size of the model it analyzes, CFGs are refined to remove nodes that are not relevant to the property being proved or are not related to intertask communication via rendezvous. Since the property in Figure 17 only refers to the events close, move, and open, and these correspond to rendezvous, the only nodes in the CFGs that are needed are those representing intertask communication or the flow of control to these communications. This refinement is safe so long as there is a weak bisimulation relationship [Milner 1989] between each original CFG and its corresponding refined CFG. Figures 20 and 21 show the CFGs for the car and controller tasks refined with respect to the property shown in Figure 17.

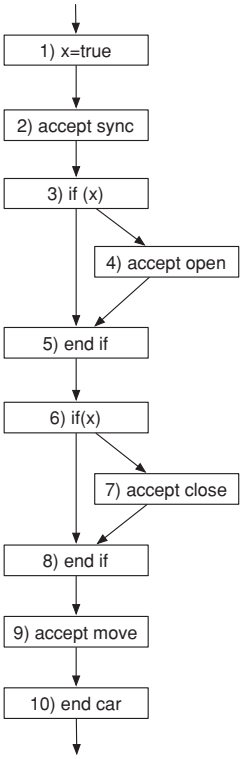

Fig. 18. CFG for task car.

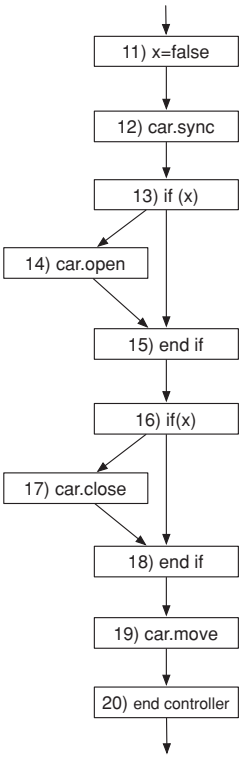

Fig. 19. CFG for task controller.

Note that the refinement algorithm also relabels with τ those nodes that do not have an event label but are necessary for control flow reasons.

As mentioned in Section 2.1, FLAVERS uses the refined CFGs to create the TFG, which represents the concurrent behavior of the whole system with respect to the events of interest. The TFG consists of a collection of CFGs with additional nodes and edges to represent intertask control flow. The TFG that is built from the CFGs in Figures 20 and 21 is shown in Figure 22. In this figure, nodes 21 and 26 are the unique initial node and final node of the TFG, respectively. To represent concurrency in Ada, extra nodes and edges are added to represent intertask communication via rendezvous. For example, node 23 and its incident edges represent the rendezvous open_doors. Additional edges are needed to represent the possible flow of control between nodes in different tasks due to task interleaving. These *May Immediately Precede* (MIP) edges are computed by the *May Happen in Parallel* (MHP) algorithm [Naumovich and Avrunin 1998] and are shown as dashed edges in Figure 22. Note that not every pair of nodes from the car and controller tasks are connected by a MIP edge. For example, the MHP algorithm can determine that nodes 9 and 20 cannot happen in parallel because the rendezvous car_move (node 25) must happen between them.

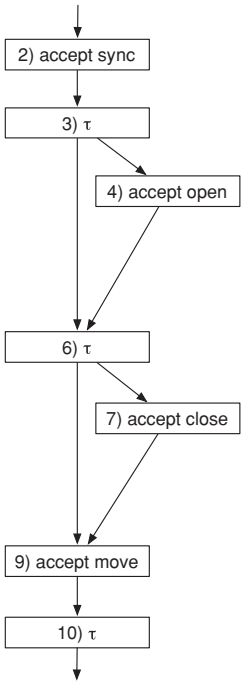

Fig. 20. Refined CFG for task car.

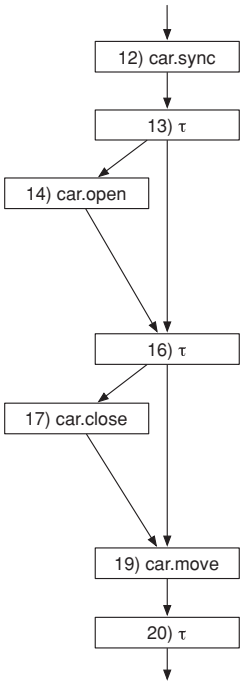

Fig. 21. Refined CFG for task controller.

A TFG is an overapproximation of the sequences of events that can occur when executing a system. Thus, every sequence of events that can occur on an execution of the system has a corresponding path in the TFG. To help keep the size of the TFG small, however, there usually are paths in the TFG that do not correspond to any actual execution of the system.

FLAVERS uses an efficient state-propagation algorithm [Olender and Osterweil 1992; Dwyer et al. 2004] to determine whether all potential behaviors of the system being analyzed are consistent with the property being verified. FLAVERS analyses are conservative, meaning FLAVERS only reports that a property holds when that property holds for all TFG paths. If FLAVERS reports that the property does not hold, this can either be because there is an execution that actually violates the property or because the property is violated on infeasible paths through the TFG.

FLAVERS would report that the property shown in Figure 17 does not hold on the TFG shown in Figure 22 because the property would be violated on the path $21 \rightarrow 2 \rightarrow 22 \rightarrow 3 \rightarrow 23 \rightarrow 6 \rightarrow 9 \rightarrow 25 \rightarrow 10 \rightarrow 26$, which corresponds to the event sequence ⟨open, move⟩. This counterexample is infeasible, however, because the variable x must be true for the edge $3 \rightarrow 23$ to be taken and false for the edge $6 \rightarrow 9$ to be taken. Since the value of x is not changed in the program between nodes 23 and 6, this path cannot occur and is thus infeasible.

This infeasible path is considered because all information related to the variable x was removed during CFG refinement. A *Variable Automaton* (VA), a constraint to track a small number of values for a variable, can be introduced to

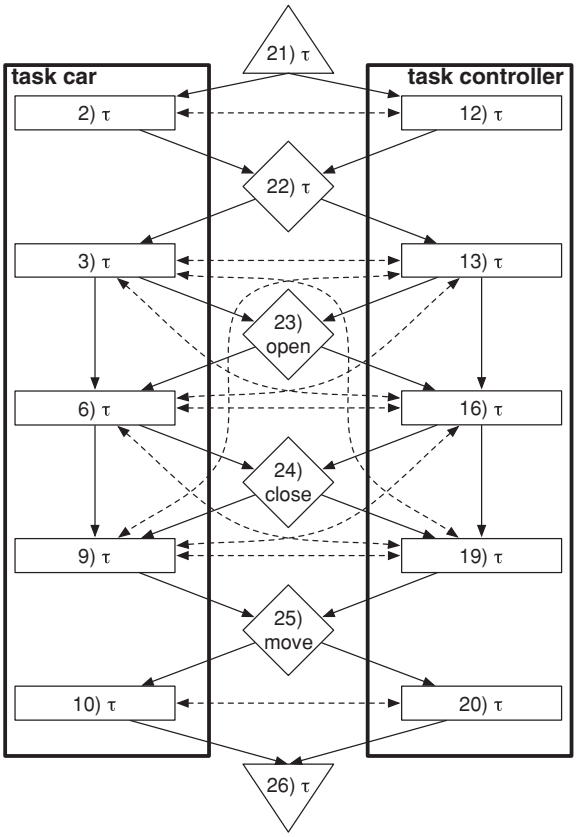

Fig. 22. TFG for the elevator system.

keep track of the values of x. In the VA for x, shown in Figure 23, events with "=" represent an assignment to x, while the events with "=" represent a test of the value of x. The three accepting states of the VA represent the unknown, true, and false values of x. The one nonaccepting state is the violation state and is entered when a path is explored that is infeasible because of an operation on x. For example, if x is known to be true and a branch is taken where the value of x is false (i.e., the event x==false occurs), then the violation state would be entered.

To make use of this VA, the TFG from Figure 22 needs to be modified so it has nodes with events corresponding to operations on x, as shown in Figure 24. In this TFG,[6] each node corresponding to a branch (nodes 3, 6, 13, and 16 from Figure 22) has been split into two nodes, one for the true branch (the nodes labeled x==true) and one for the false branch (the nodes labeled x==false). The counterexample path reported previously corresponds to the path 21 → 1 → 2 → 22 → 3a → 23 → 6b → 9 → 25 → 10 → 26 in the modified TFG. The path

[6]In our implementation, nodes 2 and 12 would be refined away, but they have been left in Figure 24 for clarity of the presentation.

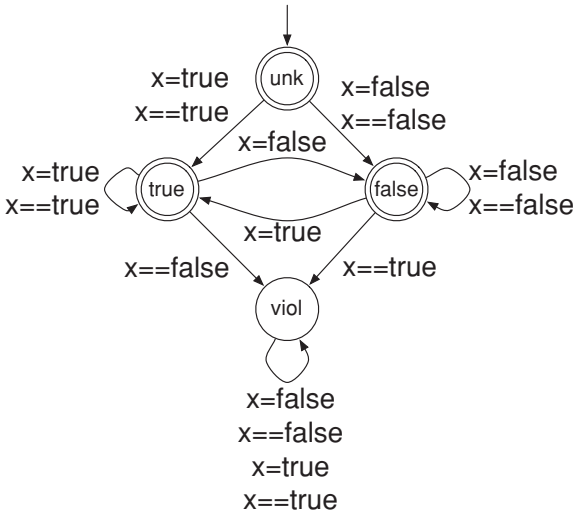

Fig. 23. Variable automaton for x.

corresponds to the event sequence $\langle x$=true, x==true, open, x==false, move$\rangle$. Because the VA for x would transition from the unknown state to the true state on node 1, remain in the true state on node 3a, and transition from the true state to the violation state on node 6b, this path would not be considered during state propagation since it leads to the violation state of a VA. Using just the VA for x, FLAVERS would report that the property shown in Figure 17 holds and that the elevator's car cannot move while its doors are open.

A.2 Implementing the Teacher in FLAVERS

Having provided a more detailed description of FLAVERS, we can now explain how a teacher for FLAVERS can be implemented.

A.2.1 *The Model.* To answer queries and conjectures, we need to build TFG models for S_1 and S_2, the two subsystems used in the assume-guarantee proof rule. The TFGs for S_1 and S_2 are similar to the TFGs that FLAVERS would normally create, but must be extended to simulate the environment in which each subsystem will execute. Thus, to model S_1 in the context of the whole system, an environment,which is modeled as a CFG, needs to be constructed to represent interactions between S_1 and S_2. Specifically, the environment for S_1 needs to have accept statements from S_2 that are called by S_1 and entry calls made by S_2 to accept statements in S_1.

Additionally, the environment for S_1 needs to contain events from S_2 that can affect the property or constraints. It is possible that an event can occur both in a task in S_1 and in a task in S_2. Events common to both S_1 and S_2 needs to be relabeled so that they can be distinguished from each other during analysis. Without this relabelling, the events in the assumption would be the same as the events in S_1. This would result in an analysis where the assumption constrains

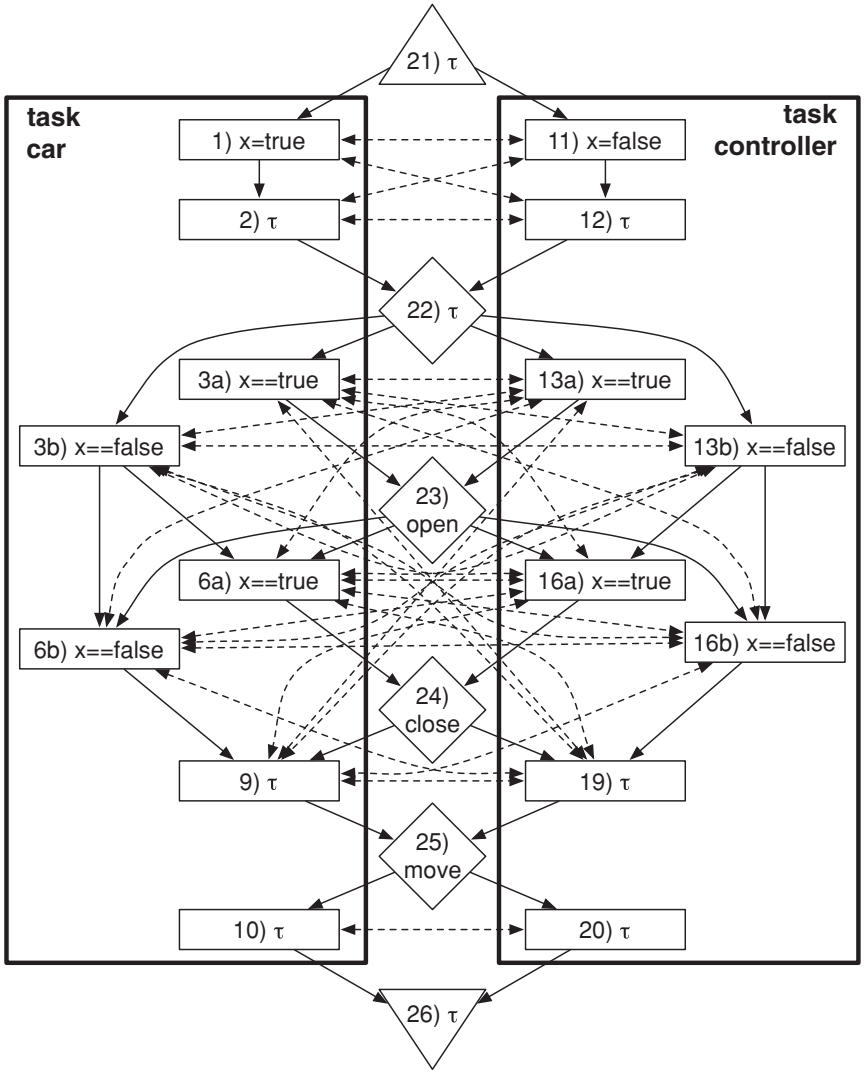

Fig. 24. TFG with events for x.

the behavior of S_1 instead of just constraining the behavior of the environment of S_1. To do this relabelling, we prefix events with "s1" and "s2." Specifically:

(1) CFGs for tasks in S_1 have common events prefixed with "s1."
(2) Common events in the environment for S_1 are prefixed with "s2."
(3) The assumption, which is represented as an FSA, is used as a constraint on the environment of S_2 and as a property that is checked on S_2. Thus, common events in the assumption are prefixed with "s2."
(4) The other FSAs (i.e., the property and other constraints) do not need to distinguish the source of the common events. Thus, common events in these

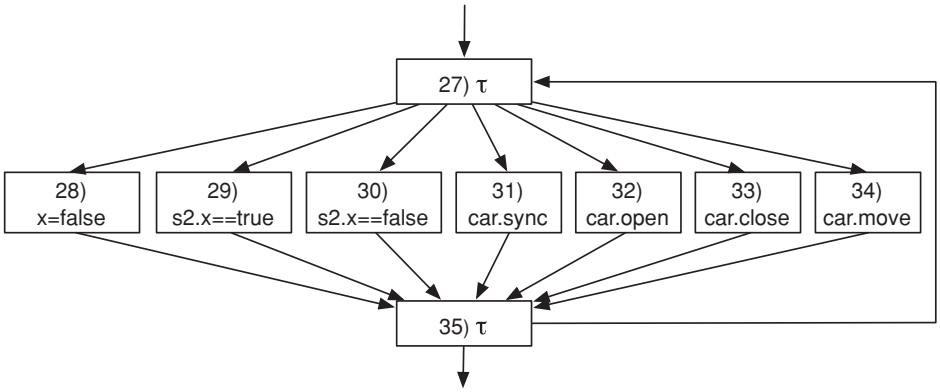

Fig. 25. Environment for S_1.

FSAs are replaced by two events, one prefixed with "s1" and one prefixed with "s2."

The environment CFG model can now be constructed to perform, in any order, zero or more accepts, entry calls, and events that can affect the property or constraints. Note, that all of the information needed to construct the environment can be gathered from the CFGs and constraints, so the environment can be automatically constructed.

Consider the elevator example with the two tasks car and controller. Let P be the property that the elevator's car cannot move while its doors are open, shown in Figure 17. Suppose that S_1 = car and S_2 = controller. In this example, tasks in S_2 make entry calls close_doors, move_car, open_doors, and sync to tasks in S_1. Tasks in S_2 accept no entry calls from tasks in S_1. The VA for the variable x has events in both the car and controller tasks, so the events from S_2, x=false, x==true, and x==false need to be in the environment for S_1. Since the events x==true and x==false also occur in S_1, these events need to be relabeled when constructing S_1 and its environment, otherwise the assumption would constrain S_1 and not just the environment of S_1. Figure 25 shows the CFG for the environment for S_1.

To build the TFG for S_1, the environment for S_1 needs to be combined with the CFGs for every task in S_1. Figure 26 shows the TFG for S_1 in the elevator example, with MIP edges removed for clarity.

A.2.2 *The Alphabet of the Assumption.* The L* algorithm learns an FSA over an alphabet Σ. To use the L* algorithm for assume-guarantee reasoning, Σ must be provided. For FLAVERS, the alphabet consists of the labels on all rendezvous[7] that occur between S_1 and S_2, and the labels on the non-τ nodes that do not correspond to rendezvous in the environment of S_1. For the elevator example, Σ_A = {close, s2.x==false, s2.x==true, x=false, move, open, sync}.

[7]This includes rendezvous that are not mentioned in the property or constraints. This is why node 37 in Figure 26 is labeled with sync instead of τ.

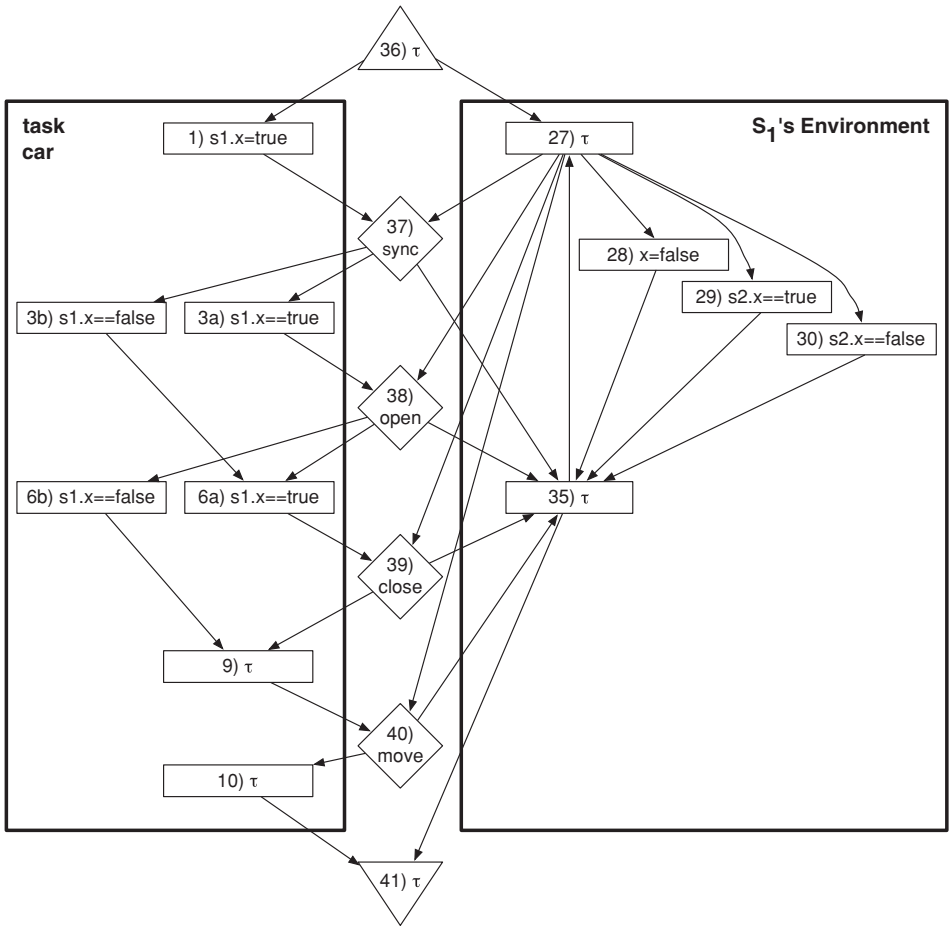

Fig. 26. TFG for S_1, without the MIP edges shown.

A.2.3 *Answering Queries.* A query posed by the L* Algorithm consists of a sequence of events from Σ^*. The teacher must answer true if this sequence is in the language being learned and false otherwise. To answer a query in FLAVERS, S_1 is represented as a TFG and the query is represented as a constraint. We then use FLAVERS to determine if the property is consistent with the TFG model as constrained by this query. If this results in a violation of the property P, then the assumption needed to make $\langle A \rangle\ S_1\ \langle P \rangle$ true should not allow the event sequence in the query and false will be returned to the L* Algorithm. Otherwise, the event sequence is permissible and true will be returned to the L* Algorithm.

To answer a query, a TFG is first constructed using the CFGs for tasks in S_1 and the CFG for the environment of S_1. The property to be checked is P. This verification uses the constraints that contain events in S_1. The CFGs, VAs, and the property P are relabeled as described previously to allow events in S_1 and S_2 to be distinguished. The query constraint is used to restrict FLAVERS to only

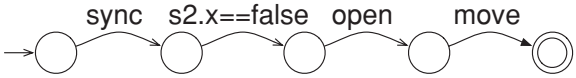

Fig. 27. Constraint for the query ⟨sync, s2.x==false, open, move⟩.

look at paths through the TFG that correspond to the event sequence specified by the query. For example, if the query were ⟨sync, s2.x==false, open, move⟩, then the constraint shown in Figure 27 would be used. State-propagation is then applied to check the property; if the property is violated then false is returned to the L* Algorithm, otherwise true is returned.

A.2.4 *Answering Conjectures.* A conjecture posed by the L* Algorithm consists of an FSA that the L* Algorithm believes accepts the language being learned. To answer a conjecture, the teacher needs to find an event sequence in the symmetric difference of the conjectured FSA and the language being learned, if such an event sequence exists. Since the conjectured FSA is the candidate assumption to be used to complete an assume-guarantee proof, it is necessary to determine if the conjectured assumption makes the two premises of the assume-guarantee proof rule true.

First, the conjectured automaton, A, is checked in Premise 1, $\langle A \rangle\ S_1\ \langle P \rangle$. To check this in FLAVERS, a TFG is constructed using the CFGs for tasks from S_1 and the CFG for the environment of S_1. The property to be checked is P. This verification uses the constraints that contain events in S_1. The CFGs, VAs, and the property P are relabeled as described previously to allow events in S_1 and S_2 to be distinguished. In addition, the assumption is used as a constraint. If this verification results in a property violation, then the counterexample returned represents an event sequence permitted by A but violating P. Thus, the conjecture is incorrect and the counterexample is returned to the L* Algorithm. If the property is not violated, then A is good enough to satisfy Premise 1 and Premise 2 can be checked.

Premise 2 states that $\langle true \rangle\ S_2\ \langle A \rangle$ should be true. To check this in FLAVERS, a TFG is constructed using the CFGs for tasks from S_2 and the CFG for the environment of S_2. Unlike the environment for S_1, which consists of events, entries (calls to rendezvous), and accepts from S_2, the environment for S_2 only consists of entries and accepts from S_1. Events from S_1 are not needed in the environment for S_2 because we not using a circular assume-guarantee rule and are only trying to ensure the S_2 satisfies the assumption independent of the behavior of S_1. Accepts and entries are needed in the environment of S_2 only so nodes corresponding to rendezvous are created correctly during TFG construction.

Rather than treat A as a constraint as was done in checking Premise 1, A is treated as a property. This verification uses the constraints that contain events only in S_2. Even though the environment does not have any events from S_1 (those prefixed with "s1"), the VAs, CFGs, and property need to be relabeled as described previously so that the alphabet of this subsystem is consistent with the alphabet of the assumption. If this verification does not result in a property violation, then both Premise 1 and Premise 2 are true, so it can be concluded

that P holds on $S_1 \parallel S_2$. If this verification results in a property violation, then the returned counterexample is examined to determine what should be done next. A query is made based on this counterexample, as described previously, to determine if P does not hold on $S_1 \parallel S_2$ or if the assumption needs to be refined.

A more detailed description of this approach can be found in Cobleigh [2007].

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

REFERENCES

ABADI, M. AND LAMPORT, L. 1995. Conjoining specifications. *ACM Trans. Prog. Lang. Syst. 17,* 3, 507–534.

AGRAWAL, R., IMIELINSKI, T., AND SWAMI, A. N. 1993. Mining association rules between sets of items in large database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 207–216.

ALUR, R., HENZINGER, T. A., MANG, F. Y. C., QADEER, S., RAJAMANI, S. K., AND TASIRAN, S. 1998. MOCHA: Modularity in model checking. In *Proceedings of the 10th International Conference on Computer-Aided Verification*. A. J. Hu and M. Y. Vardi, Eds. Lecture Notes in Computer Science, vol. 1427. 521–525.

ALUR, R., MADHUSUDAN, P., AND NAM, W. 2005. Symbolic compositional verification by learning assumptions. In *Proceedings of the 17th International Conference on Computer-Aided Verification*. K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. 548–562.

ANDREWS, T., QADEER, S., RAJAMANI, S. K., AND XIE, Y. 2004. Zing: Exploiting program structure for model checking concurrent software. In *Proceedings of the 15th International Conference on Concurrency Theory*. P. Gardner and N. Yoshida, Eds. Lecture Notes in Computer Science, vol. 3170. 1–15.

ANGLUIN, D. 1987. Learning regular sets from queries and counterexamples. *Inform. Computa. 75,* 2, 87–106.

AVRUNIN, G. S., CORBETT, J. C., AND DWYER, M. B. 2000. Benchmarking finite-state verifiers. *Int. J. Softw. Tools Tech. Trans. 2,* 4, 317–320.

AVRUNIN, G. S., CORBETT, J. C., DWYER, M. B., PĂSĂREANU, C. S., AND SIEGEL, S. F. 1999. Comparing finite-state verification techniques for concurrent software. Tech Rep. 99-69, Department of Computer Science, University of Massachusetts.

BARRINGER, H., GIANNAKOPOULOU, D., AND PĂSĂREANU, C. S. 2003. Proof rules for automated compositional verification through learning. In *Proceedings of the 2nd Workshop on Specification and Verification of Component-Based Systems*. 14–21.

BERRY, G. AND BOUDOL, G. 1992. The chemical abstract machine. *Theor. Comput. Sci. 96,* 1, 217–248.

BRAT, G., DENNEY, E., GIANNAKOPOULOU, D., FRANK, J., AND JÓNSSON, A. 2006. Verification of autonomous systems for space applications. In *Proceedings of the IEEE Aerospace Conference*.

CHAKI, S., CLARKE, E., GIANNAKOPOULOU, D., AND PĂSĂREANU, C. S. 2004. Abstraction and assume-guarantee reasoning for automated software verification. Tech. Rep. 05.02, Research Institute for Advanced Computer Science.

CHAKI, S., CLARKE, E. M., SINHA, N., AND THATI, P. 2005. Automated assume-guarantee reasoning for simulation conformance. In *Proceedings of the 17th International Conference on Computer-Aided Verification*. K. Etessami and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 3576. 534–547.

CHAKI, S. AND STRICHMAN, O. 2007. Optimized L*-based assume-guarantee reasoning. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Sytems*. O. Grumberg and M. Huth, Eds. Lecture Notes in Computer Science, vol. 4424. 276–291.

CHATLEY, R., EISENBACH, S., AND MAGEE, J. 2004. MagicBeans: a platform for deploying plugin components. In *Proceedings of the 2nd International Working Conference on Component Development*. W. Emmerich and A. L. Wolf, Eds. Lecture Notes in Computer Science, vol. 3083. 97–112.

CHEUNG, S.-C. AND KRAMER, J. 1996. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Engin. Method. 5,* 4, 334–377.

CIMATTI, A., CLARKE, E. M., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R., AND TACCHELLA, A. 2002. NuSMV 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification*. E. Brinksma and K. G. Larsen, Eds. Lecture Notes in Computer Science, vol. 2404. 359–364.

CLARKE, E. M., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2000. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer-Aided Verification*. E. A. Emerson and A. P. Sistla, Eds. Lecture Notes in Computer Science, vol. 1855. 154–169.

COBLEIGH, J. M. 2007. Automating and evaluating assume-guarantee reasoning. Ph.D. thesis, University of Massachusetts, Amherst.

COBLEIGH, J. M., GIANNAKOPOULOU, D., AND PĂSĂREANU, C. S. 2003. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. H. Garavel and J. Hatcliff, Eds. Lecture Notes in Computer Science, vol. 2619. 331–346.

CORBETT, J. C. AND AVRUNIN, G. S. 1995. Using integer programming to verify general safety and liveness properties. *Form. Meth. Syst. Des. 6,* 1, 97–123.

DE LA RIVA, C. AND TUYA, J. 2004. Modular model checking of software specifications with simultaneous environment generation. In *Proceedings of the 2nd International Conference on Automated Technology for Verification and Analysis*. F. Wang, Ed. Lecture Notes in Computer Science, vol. 3299. 369–383.

DE LA RIVA, C., TUYA, J., AND DE DIEGO, J. R. 2001. Modular model checking of SA/RT models using association rules. In *Proceedings of the 1st International Workshop on Model-Based Requirements Engineering*. 61–68.

DWYER, M. B., CLARKE, L. A., COBLEIGH, J. M., AND NAUMOVICH, G. 2004. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. Softw. Engin. Method. 13,* 4, 359–430.

FLANAGAN, C., FREUND, S. N., QADEER, S., AND SESHIA, S. A. 2005. Modular verification of multithreaded programs. *Theor. Comput. Sci. 338,* 1–3, 153–183.

FLANAGAN, C. AND QADEER, S. 2003. Thread-modular model checking. In *Proceedings of the 10th SPIN Workshop*. T. Ball and S. K. Rajamani, Eds. Lecture Notes in Computer Science, vol. 2648. 213–224.

FOURNET, C., HOARE, T., RAJAMANI, S. K., AND REHOF, J. 2004. Stuck-free conformance. In *Proceedings of the 16th International Conference on Computer-Aided Verification*. R. Alur and D. Peled, Eds. Lecture Notes in Computer Science, vol. 3114. 242–254.

GHEORGHIU, M., GIANNAKOPOULOU, D., AND PĂSĂREANU., C. S. 2007. Refining interface alphabets for compositional verification. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. O. Grumberg and M. Huth, Eds. Lecture Notes in Computer Science, vol. 4424. 292–307.

GIANNAKOPOULOU, D., KRAMER, J., AND CHEUNG, S.-C. 1999. Behaviour analysis of distributed systems using the Tracta approach. *Automat. Softw. Engin. 6,* 1, 7–35.

GIANNAKOPOULOU, D. AND PĂSĂREANU, C. S. 2005. Learning-based assume-guarantee verification. In *Proceedings of the 12th SPIN Workshop*. P. Godefroid, Ed. Lecture Notes in Computer Science, vol. 3639. 282–287.

GIANNAKOPOULOU, D., PĂSĂREANU, C. S., AND BARRINGER, H. 2002. Assumption generation for software component verification. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*. 3–12.

GROCE, A., PELED, D., AND YANNAKAKIS, M. 2002. Adaptive model checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. J.-P. Katoen and P. Stevens, Eds. Lecture Notes in Computer Science, vol. 2280. 357–370.

GRUMBERG, O. AND LONG, D. E. 1994. Model checking and modular verification. *ACM Trans. Program. Lang. Syst. 16,* 3, 843–871.

HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTUL-TRAURING, A., AND TRAKHTENBROT, M. 1990. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. Softw. Engin. 16,* 4, 403–414.

HELMBOLD, D. AND LUCKHAM, D. 1985. Debugging Ada tasking programs. *IEEE Softw. 2,* 2, 47–57.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND QADEER, S. 2003. Thread-modular abstraction refinement. In *Proceedings of the 15th International Conference on Computer-Aided Verification*. W. A. Hunt, Jr. and F. Somenzi, Eds. Lecture Notes in Computer Science, vol. 2725. 262–274.

HENZINGER, T. A., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*. 58–70.

HENZINGER, T. A., QADEER, S., AND RAJAMANI, S. K. 1998. You assume, we guarantee: Methodology and case studies. In *Proceedings of the 10th International Conference on Computer-Aided Verification*. A. J. Hu and M. Y. Vardi, Eds. Lecture Notes in Computer Science, vol. 1427. 440–451.

HOARE, C. A. R. 1969. An axiomatic basis for computer programming. *Commu. ACM 12,* 10, 576–580.

INVERARDI, P., WOLF, A. L., AND YANKELEVICH, D. 2000. Static checking of system behaviors using derived component assumptions. *ACM Trans. Softw. Engin. Method. 9,* 3, 239–272.

JEFFORDS, R. D. AND HEITMEYER, C. L. 2003. A strategy for efficiently verifying requirements. In *Proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 28–37.

JONES, C. B. 1983. Specification and design of (parallel) programs. In *Proceedings of the IFIP 9th World Congress*, R. Mason, Ed. IFIP: North Holland, 321–332.

KELLER, R. K., CAMERON, M., TAYLOR, R. N., AND TROUP, D. B. 1991. User interface development and software environments: The Chiron-1 system. In *Proceedings of the 13th International Conference on Software Engineering*. 208–218.

MAGEE, J. AND KRAMER, J. 1999. *Concurrency: State Models & Java Programs*. John Wiley & Sons.

MCMILLAN, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

MCMILLAN, K. L. 1998. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proceedings of the 10th International Conference on Computer-Aided Verification*. A. J. Hu and M. Y. Vardi. Eds. Lecture Notes in Computer Science, vol. 1427. 110–121.

MILNER, R. 1989. *Communication and Concurrency*. Prentice-Hall.

NAUMOVICH, G. AND AVRUNIN, G. S. 1998. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 24–34.

OLENDER, K. M. AND OSTERWEIL, L. J. 1992. Interprocedural static analysis of sequencing constraints. *ACM Trans. Softw. Engin. Method. 1,* 1, 21–52.

PĂSĂREANU, C. S., DWYER, M. B., AND HUTH, M. 1999. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*. D. Dams, R. Gerth, S. Leue, and M. Massink, Eds. Lecture Notes in Computer Science, vol. 1680. 168–183.

PATIL, S. S. 1971. Limitations and capabilities of Dijktra's semaphore primitives for coordination among processes. *Computational Structures Group Memo 57, Project MAC*.

PETERSON, G. L. 1981. Myths about the mutual exclusion problem. *Inform. Process. Lett. 12,* 3 (June), 115–116.

PNUELI, A. 1984. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, K. R. Apt, Ed. NATO ASI, vol. 13. Springer-Verlag, 123–144.

RIVEST, R. L. AND SCHAPIRE, R. E. 1993. Inference of finite automata using homing sequences. *Inform. Computa. 103,* 2, 299–347.

SHUREK, G. AND GRUMBERG, O. 1990. The modular framework of computer-aided verification. In *Proceedings of the 2nd International Workshop on Computer-Aided Verification*. E. M. Clarke and R. P. Kurshan, Eds. Lecture Notes in Computer Science, vol. 531. 214–223.

SIEGEL, S. F. AND AVRUNIN, G. S. 2002. Improving the precision of INCA by eliminating solutions with spurious cycles. *IEEE Trans. Softw. Engin. 28,* 2, 115–128.

TAYLOR, R. N., BELZ, F. C., CLARKE, L. A., OSTERWEIL, L. J., SELBY, R. W., WILEDEN, J. C., WOLF, A. L., AND YOUNG, M. 1988. Foundations for the Arcadia environment architecture. In *Proceedings of*

the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. 1–13.

TKACHUK, O., DWYER, M. B., AND PĂSĂREANU, C. 2003. Automated environment generation for software model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*. 116–129.

YEH, W. J. AND YOUNG, M. 1991. Compositional reachability analysis using process algebra. In *Proceedings of the 1991 Symposium on Testing, Analysis, and Verification*. 49–59.

Received April 2007; revised August 2007; accepted October 2007