

Breaking Up is Hard to Do: An Investigation of Decomposition for Assume-Guarantee Reasoning*

Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke
Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA

jacobleig@cs.umass.edu, avrunin@cs.umass.edu, clarke@cs.umass.edu

ABSTRACT

Finite-state verification techniques are often hampered by the state-explosion problem. One proposed approach for addressing this problem is assume-guarantee reasoning. Using recent advances in assume-guarantee reasoning that automatically generate 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, we evaluated the decompositions as the systems were scaled. In only a few cases could assume-guarantee reasoning verify properties on larger systems than monolithic verification and, in these cases, assume-guarantee reasoning could only verify these properties on systems a few sizes larger than monolithic verification. This discouraging result, although preliminary, raises doubts about the usefulness of assume-guarantee reasoning.

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

General Terms: Verification, Experimentation

Keywords: Assume-guarantee reasoning

1. INTRODUCTION

Finite-state verification techniques are being developed to verify behavioral properties of distributed and concurrent systems. One of the major problems that these techniques must address is the state-explosion problem, where the number of reachable states to be explored is exponential in the number of concurrent tasks in a

*This research was partially supported by the National Science Foundation under grants CCR-0205575 and CCF-0427071, 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSA'06, July 17–20, 2006, Portland, Maine, USA.

Copyright 2006 ACM 1-59593-263-1/06/0007 ...\$5.00.

system. Compositional reasoning techniques have been proposed as one way to address the state-explosion problem. One of the most frequently advocated compositional reasoning techniques is assume-guarantee reasoning [28, 35] in which a verification problem is represented as a triple, $\langle A \rangle S \langle P \rangle$, where

- S is the subsystem being analyzed,
- P is the property to be verified, and
- A is an assumption about the environment in which S is used.

The formula $\langle A \rangle S \langle P \rangle$ is true if, whenever S is used in an environment satisfying A , then the property P must hold. Consider a system S decomposed into two subsystems, S_1 and S_2 (which may then be further decomposed). To verify that a property P holds on the system made up of S_1 and S_2 , denoted $S_1 \parallel S_2$, the following is the simplest assume-guarantee rule that can be used:

$$\frac{\text{Premise 1: } \langle A \rangle S_1 \langle P \rangle \quad \text{Premise 2: } \langle \text{true} \rangle S_2 \langle A \rangle}{\langle \text{true} \rangle S_1 \parallel S_2 \langle P \rangle}$$

By checking these two premises, a property can be verified on $S_1 \parallel S_2$ without ever having to examine $S_1 \parallel S_2$.

There are several issues that make using the above 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 subsystems. Deciding how to partition the subsystems into S_1 and S_2 is not easy and can have a significant impact on the time and memory needed for verification. 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 completes the assume-guarantee proof. Because of these difficulties, it had not been practical to undertake an empirical evaluation of this approach, although several case studies have been reported (e.g., [18, 25]).

There has been recent work on automatically computing assumptions for compositional analysis [1, 5, 6, 11, 21, 24]. This 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 the algorithm presented in [11], we undertook such a study to evaluate the effectiveness of assume-guarantee reasoning.

We initially undertook this 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 implemented this automated assume-guarantee reasoning algorithm for two verifiers, FLAVERS [15] and LTSA [30]. We began by looking at several examples using FLAVERS, but the results of these experiments were not as promising as the results seen in [11], which used LTSA. Although the two tools use different models and verification methods, this was surprising to us. As a result, we translated

our examples into LTSA to see if our choice of tool affected our results. In the study reported here, we applied both tools to a small set of scalable systems and properties. For each property of each system, we examined *all* of the ways to partition the subsystems of that system into S_1 and S_2 at the smallest system size to find the best decomposition, that is, the decomposition for which assume-guarantee reasoning explores the fewest states. Because examining all decompositions for each of the larger system sizes is infeasible due to the time cost, we generalized the best decompositions found for smaller system sizes and used those generalizations for assume-guarantee reasoning at larger sizes. To evaluate these generalized decompositions, however, we did explore all two-way decompositions for a few larger sizes, although we were not able to find the best decomposition in all cases because of the time required. In total, we examined over 43,000 two-way decompositions and used over 1.43 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 reasoning¹ than are explored using monolithic verification. If we restrict our attention to the best decomposition for each example, then we found that in about half of the examples 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, compositional analysis often explores fewer states than monolithic verification. This memory savings, however, was rarely enough to increase the size of the systems that could be verified beyond what could be done with monolithic verification.

Section 2 provides some background information about the finite-state verifiers and the automated assume-guarantee algorithm we used. In Section 3 we describe our experimental methodology and results. We end by discussing related work and conclusions.

2. BACKGROUND

This section gives a brief description of the two verifiers used in our experiments, FLAVERS and LTSA. It also briefly describes the L* algorithm and how it can be used for automated assume-guarantee reasoning.

2.1 FLAVERS

FLAVERS (Flow Analysis for Verification of Systems) is a finite-state verifier that can prove user-specified properties of sequential and concurrent systems [15]. 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 is translated 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 corresponds to the 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 concurrent systems. The TFG consists of a collection of CFGs with additional nodes and edges to represent intertask control flow. A CFG, and

¹When discussing the number of states explored by assume-guarantee reasoning, we mean the maximum number of states explored by a verifier when computing an assumption or checking the two premises. Since assume-guarantee reasoning is intended to reduce the impact of the state explosion problem, the maximum number of states explored at any one time seems to be the appropriate measure rather than the sum of the number of states explored.

thus a TFG, over-approximates the sequences of events that can occur when executing a system.

FLAVERS uses an efficient state-propagation algorithm 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 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. These infeasible paths do not correspond to any possible execution of the system but are an artifact of the imprecision of the model. The analyst can introduce *feasibility 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 feasibility constraints and observe the analysis results several times before determining whether or not a property holds. Feasibility constraints give analysts some control over the analysis process by letting them determine exactly what parts of a system need to be modeled in order to prove a property.

The FLAVERS state-propagation algorithm has worst-case complexity that is $O(N^2 \cdot Q)$, where N is the number of nodes in the TFG, and Q is the product of the number of states in the property and all constraints. Experimental evidence shows that the performance of FLAVERS is often sub-cubic in the size of the system [15] and that the performance of FLAVERS is good when compared to other finite-state verifiers [3,4].

2.2 LTSA

LTSA (Labeled Transition Systems Analyzer) is a finite-state verifier that can prove user-specified properties of sequential and concurrent systems [30]. LTSA can check both safety and liveness properties, but the assume-guarantee algorithm we use can only handle safety properties. Safety properties in LTSA are specified as FSAs where every state except one is an accepting state. This single non-accepting state must be a trap state, meaning all transitions that leave it must be self-loop transitions. 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. LTSs are written in FSP, a process-algebra style notation.

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 (\parallel). 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 supports *Compositional Reachability Analysis* (CRA) of a software system based on its hierarchical structure. CRA incrementally computes and abstracts the behavior of composite components using the architecture of the system as a guide to the order to perform the composition [19]. While CRA can reduce the cost of verification, state explosion can still occur and result in the cost of verification being exponential in the number of concurrent tasks in the system.

2.3 Using the L* Algorithm for Automated Assume-guarantee Reasoning

The L* algorithm was originally developed by Angluin [2] and later improved by Rivest and Schapire [36]. In this work, we

use Rivest and Schapire’s version of the L^* algorithm. The L^* algorithm learns an FSA for an unknown regular language over an alphabet Σ through its interactions 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, and the questions it poses are based on the answers it received to previously asked questions. Due to space limitations, we cannot describe the L^* algorithm, but do provide a high-level description of 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^* , and 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 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 returns *true* if the conjecture is correct. Otherwise, the teacher returns *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.

First, the conjectured automaton, A , is checked in Premise 1, $\langle A \rangle S_1 \langle P \rangle$. 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 , but violating 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 $S_1 \parallel S_2$. 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 that is done 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 will result in a property violation when S_2 interacts with S_1 , so it can be concluded that P does not hold on $S_1 \parallel S_2$. 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 restricting the behavior of S_2 unnecessarily. The counterexample is then returned to the L^* algorithm in response to the conjecture.

2.3.3 Complexity and Correctness

It is shown in [11] that this approach to assume-guarantee reasoning terminates and correctly determines whether or not $S_1 \parallel S_2$

satisfies P . The description of the teacher we provide is at a high level, but works for both FLAVERS and LTSA. Differences in the models used by FLAVERS and LTSA necessitate differences in the implementation of their teachers. The teacher for LTSA was described in [11] and we describe the teacher for FLAVERS in Appendix B.

Using Rivest and Schapire’s version of the L^* algorithm, $l - 1$ conjectures and $O(kl^2 + l \log m)$ queries are needed 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.

While 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 ($\langle A \rangle S_1 \langle P \rangle$) true, it might learn an assumption that is smaller than S_2 but that allows more behavior than S_2 . This could result in the verification of Premise 1 using more memory than monolithic verification. Similarly, 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 ($\langle true \rangle S_2 \langle A \rangle$) may use more memory than monolithic verification.

3. METHODOLOGY AND RESULTS

Our goal was to try to gain a sense of whether or not the automated assume-guarantee reasoning algorithm presented in [11] provides an advantage over monolithic verification. There are several different ways that this technique could provide such an advantage:

1. Does this technique use less memory than monolithic verification?
2. Does this technique use less time than monolithic verification?
3. Can this technique verify properties on larger systems than monolithic verification?

Since finite-state verification techniques are more often limited by memory than by time, we focused our study on points 1 and 3 rather than point 2.

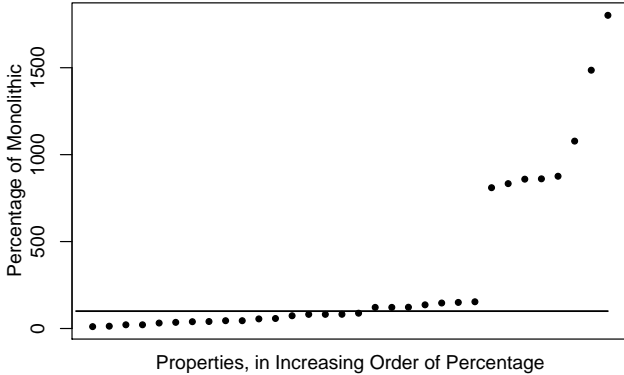
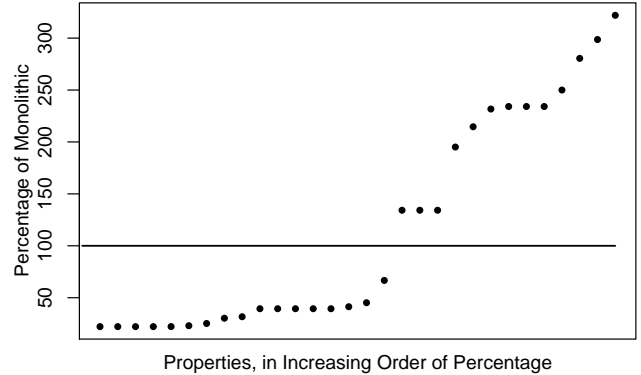
To evaluate the usefulness of this automated assume-guarantee reasoning technique, we tried to verify properties that were known to hold on a small set of scalable systems: the Chiron user interface system [29] (both the single and the multiple dispatcher versions as described in [4]), the Gas Station problem [23], Peterson’s mutual exclusion protocol [34], the Relay problem [37], and the Smokers problem [33]. These systems were specified in Ada and use rendezvous for intertask communication. Peterson’s mutual exclusion protocol also uses shared variables for intertask communication. Except for Peterson’s mutual exclusion protocol, 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. For LTSA, INCA [12] was used to translate the Ada systems into FSAs, which are easily translated into LTSS. We used the version of FLAVERS that directly accepts Ada programs.

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. While 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.

Since FLAVERS uses constraints to control the amount of precision in a verification, we used a minimal set of constraints when

Table 1: Number of two-way decompositions examined for systems of size 2

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

**Figure 1: Memory Used by the Best Decomposition of Size 2 for FLAVERS****Figure 2: Memory Used by the Best Decomposition of Size 2 for LTSA**

verifying properties monolithically and compositionally.² We did not use the most recent version of LTSA, which is based on plugins [7], because the plugin interface does not provide direct access to the LTSs. An implementation of this automated assume-guarantee reasoning technique exists for the plugin version of LTSA [20], but verification takes significantly longer because all LTSs must be created by writing appropriate FSP, necessitating re-parsing the entire model during each query and conjecture, even for the parts of the model that do not change between different queries and conjectures.

Since the scalable systems we looked at had more than two subsystems, we needed to partition those subsystems into S_1 and S_2 to use the assume-guarantee rule presented earlier. For both FLAVERS and LTSA we considered a task to be an indivisible subsystem. We wanted to find a two-way decomposition where neither S_1 nor S_2 is empty, on which assume-guarantee reasoning used the least amount of memory.

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

²A minimal set of constraints is one such that removal of any constraint causes FLAVERS to report that the property may not hold. 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 for FLAVERS. 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. These minimal sets were found using the process described in [15].

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 count the number of smokers.

3.1 Does Assume-guarantee Reasoning Save Memory at Small System Sizes?

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. While the data structures used by the L^* algorithm and the artifacts created by the verifiers (e.g. TFGs and FSAs in FLAVERS, LTSs in LTSA) do use memory, we did not count them when determining memory usage since the amount of memory needed to store them is small when compared to the amount of memory needed to store the states explored when queries and conjectures are answered. We say one decomposition is 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. On some properties, assume-guarantee reasoning using the L^* algorithm explored more states than would have been explored had an assumption been supplied and just $\langle A \rangle S_1 \langle P \rangle$ and $\langle true \rangle S_2 \langle A \rangle$ checked. We count this additional overhead when determining the amount of memory used by assume-guarantee reasoning and discuss this issue in Section 3.5. Similarly, we determine the amount of memory used by monolithic verification by looking at the number of states explored during verification and do not consider the amount of memory needed to hold the artifacts created by the verifiers.

Initially, we selected several decompositions for each property of each system at size 2 based on our understanding of the system and of assume-guarantee reasoning. We expected that in many

Table 2: Performance of the best decompositions for size 2 and the generalized decompositions compared to monolithic verification

Tool	Total number of properties	Best decomp. at size 2 is better than mono.	Percentage	Generalized decomp. is better than mono.	Percentage
FLAVERS	32	17	53.1%	18	56.3%
LTSA	30	17	56.7%	5	16.7%

cases assume-guarantee reasoning would save memory over monolithic verification, although we knew that this might not always be the case. In our experiments, we were surprised to discover that in over half of our examples the decompositions we selected did not use less memory than monolithic verification. Then, for each property of each system with each verifier, we examined *all* two-way decompositions to find the best decomposition with respect to memory. For some of the examples where the decompositions we selected did not save memory, our exhaustive search found decompositions that use less memory than monolithic verification, showing that our intuition about selecting appropriate decompositions was not good. For other examples, however, every decomposition uses more memory than monolithic verification.

Table 1 lists, for FLAVERS and LTSA, the number of properties for each system,³ the number of two-way decompositions⁴ examined for each property when each system is size 2, and the total number of decompositions examined on each system when that system is size 2.

Figures 1 and 2 show, for FLAVERS and LTSA respectively, the amount of memory used by the best decomposition at size 2 as a percentage of the amount of memory used by monolithic verification. For reference, a line at 100% has been drawn. Points below this line represent properties on which the best decomposition is better than monolithic verification.

For FLAVERS, the best decomposition is better than monolithic verification on 17 of the 32 properties. For these 17 properties, on average the best decomposition uses 48.4% of the memory used by monolithic verification. For the 15 properties where the best decomposition is worse than monolithic verification, on average the best decomposition uses 637.1% of the memory used by monolithic verification.

For LTSA, the best decomposition is better than monolithic verification on 17 of the 30 properties with LTSA.⁵ For these 17 properties, on average the best decomposition uses 33.6% of the memory used by monolithic verification. For the 13 properties where the best decomposition is worse than monolithic verification, on average the best decomposition uses 222.9% of the memory used by monolithic verification.

It is important to note that the vast majority of decompositions are not better than monolithic verification. Even for the properties where the best decomposition is better than monolithic verification, most of the decompositions we examined for those properties are

³There is one fewer Chiron property for LTSA than for FLAVERS. The events needed to express that property are removed from the model when LTSA constructs the LTSs. Since this property states that those events cannot occur, LTSA proves this property holds during model construction, making verification unnecessary.

⁴Note that the number of two-way decompositions examined for each property are always two fewer than a power of two because the two-way decompositions that put all of the subsystems into either S_1 or S_2 are not checked.

⁵These are not the same 17 properties as for FLAVERS. There are a total of 13 properties for which the assume-guarantee approach is better than monolithic verification for both FLAVERS and LTSA.

not better than monolithic verification. Thus, randomly selecting decompositions would likely not yield a decomposition better than monolithic verification. Although it might be possible to develop heuristics to aid in decomposing a system, in our experiments we did not see any patterns in the decompositions that saved memory that could be used as the basis for heuristics.

Although assume-guarantee reasoning using learned assumptions could save memory in only about half of our examples when the best decomposition is used and finding the decompositions to make assume-guarantee reasoning most effective was expensive, the overall approach was not too onerous. On average, it required about two minutes to examine one decomposition with FLAVERS and about half a minute to examine one decomposition with LTSA. It is infeasible to evaluate all two-way decompositions for larger system sizes, however, because the number of decompositions to be evaluated increases exponentially and the cost of evaluating each decomposition increases as well. In the next section we examine whether or not the best decomposition for size 2 can be used to help find a decomposition that can save memory compared to monolithic verification at larger system sizes.

3.2 Does Assume-guarantee Reasoning Save Memory at Larger System Sizes?

While the cost to find the best decomposition for each property at size 2 was not too great, it is infeasible to evaluate all two-way decompositions for larger system sizes. We have several instances where evaluating a single decomposition on a system of size 4 took over 1 month. Thus, if memory was a concern in verifying a specific system and it was important to verify it for a larger size, a reasonable approach might be to examine all decompositions for a small system size and then generalize the best decomposition for that small system size to a larger system size. We used this approach to evaluate the memory usage of assume-guarantee reasoning for larger system sizes, and our algorithm for generalizing decomposition from the best decomposition for size 2 is described in Appendix A.

Table 2 gives the performance of the best decomposition at size 2 compared to monolithic verification. It also gives the performance of the generalized decomposition compared to monolithic verification at the largest system size that such a comparison could be made. This size varies from property to property depending on when compositional analysis and monolithic verification run out of memory.

With FLAVERS, if the best decomposition at size 2 is better than monolithic verification, the associated generalized decomposition is usually better than monolithic verification. While there were 17 properties on which the best decomposition at size 2 is better than monolithic verification, there were 18 properties on which the generalized decomposition is better than monolithic verification. There was 1 property on which the best decomposition at size 2 is better than monolithic verification, but the generalized decomposition is not better than monolithic verification. There were also 2 properties on which the best decomposition at size 2 is worse than mono-

Table 3: Generalized Decompositions Compared to Monolithic Verification with respect to Scaling

		FLAVERS		LTSA	
		Number of Properties	Percentage	Number of Properties	Percentage
Potential Success	1. Generalized can scale farther than monolithic	6	18.8%	0	0.0%
	2. Don't know, but generalized appears better than monolithic	7	21.9%	5	16.7%
	Subtotal	13	40.6%	5	16.7%
Likely Failure	3. Generalized cannot scale farther than monolithic	13	40.6%	0	0.0%
	4. Don't know, but generalized appears worse than monolithic	2	6.3%	25	83.3%
	5. Monolithic scales well, so generalized unlikely to be of much use	4	12.5%	0	0.0%
	Subtotal	19	59.4%	25	83.3%
Total		32	100.0%	30	100.0%

lithic verification, but the generalized decomposition is better than monolithic verification.

With LTSA, of the 17 properties 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.

In summary, with FLAVERS the best decomposition at size 2 is better than monolithic verification on 17 of the 32 examples, or about 53% of the time, and the generalized decomposition is better than monolithic verification on 18 of the 32 examples, or about 56% of the time. With LTSA the best decomposition at size 2 is better than monolithic verification on 17 of the 30 examples, or about 56% of the time, and the generalized decomposition is better than monolithic verification on 5 of the 30 examples, or 1/6 of the time. Thus, the automated assume-guarantee reasoning technique we used was able to save memory on larger size systems in a bit more than half the cases with FLAVERS and about 1/6 of the cases with LTSA. This memory savings, however, is probably not of much value unless it is significant enough to allow properties to be verified that could not be verified monolithically. In the next section, we examine whether or not the generalized decompositions allowed us to verify properties on systems larger than could be verified monolithically.

3.3 Can Assume-guarantee Reasoning Verify Properties of Larger Systems than Monolithic Verification Can?

We tried to determine, for each property, whether or not compositional analysis using the generalized decompositions, as described in Appendix A, would allow us to verify that property on larger systems sizes than monolithic verification.

Unfortunately, we were not able to determine an answer to this question for each property in our study. There were some systems on which we were unable to build models for larger system sizes. As a result, we were unable to definitively determine whether or not compositional analysis using the generalized decompositions would allow us to verify properties on larger system sizes than monolithic verification would. For example, the language processing toolkit used by FLAVERS⁶ [38] was unable to generate models

⁶This toolkit is very old and not easily modifiable. We are working on building models for FLAVERS from Java programs using Bandera [13] and, thus, expect to remove some of the limitations of our old language processing toolkit.

Chiron systems larger than size 6. After running these experiments, we assigned each property to one of five categories:

1. Compositional analysis can verify a larger system than monolithic verification.
2. It is unknown if compositional analysis can verify a larger system than monolithic verification. We consider it likely, however, because compositional analysis is substantially better than monolithic verification for the largest system size such a comparison could be made.
3. Compositional analysis cannot verify a larger system than monolithic verification.
4. It is unknown if compositional analysis can verify a larger system than monolithic verification. We consider it unlikely, however, because compositional analysis is worse than monolithic verification for the largest system size such a comparison could be made.
5. Compositional analysis is better than monolithic verification, but monolithic verification can verify the property on systems with sizes 45 or more. While compositional analysis might be able to verify a larger system, we think verifying these properties on larger systems will not be of much use.⁷

Table 3 shows the number of properties in each category for FLAVERS and LTSA. We consider using generalized decompositions to be a potential success in verifying a larger system than monolithic verification if the property is in category 1 or 2. We consider using generalized decomposition to be a likely failure in verifying a larger system than monolithic verification if the property is in category 3, 4, or 5. This yields a potential success rate of about 40% for FLAVERS and about 16% for LTSA. Note that this rate is what we believe to be the upper bound of the success rate. By looking at just the properties where we could demonstrate that compositional analysis could scale farther, meaning those in category 1, we obtain the lower bound of the success rate: about 18% for FLAVERS and 0% for LTSA.

While we could demonstrate that assume-guarantee reasoning could scale farther than monolithic verification on six properties, it is also important to look at how much farther assume-guarantee reasoning could scale. On five of these six properties, compositional

⁷The somewhat arbitrary cutoff of 45 represents a substantial size for the systems under consideration. All of the properties we examined that could be verified on systems larger than size 10 could be verified on systems with size 45.

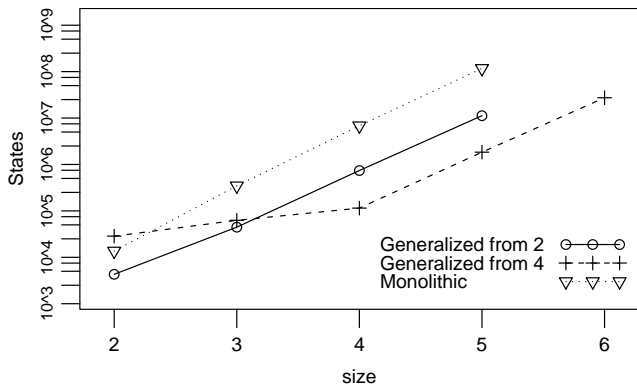


Figure 3: States Explored on the “Mutual Exclusion” Property of Gas Station with FLAVERS

analysis could verify the property on a system one size larger, but not two sizes larger, than monolithic verification could.⁸ On the sixth property, compositional analysis could verify the property on a system two sizes larger, but not three sizes larger, than monolithic verification could. In addition, there was one property, counted in category 5, where compositional analysis could verify the system at least three sizes larger than monolithic verification could. This allowed us to increase the size of the system that could be verified from 47 to 50. Since monolithic verification could scale to size 47, a fairly large system size, we do not believe verifying the system on size 50 is particularly useful and do not count this example as a success. While there were six properties where we could demonstrate that assume-guarantee reasoning could scale farther than monolithic verification, compositional analysis could verify these properties on systems only slightly larger than the size of the system on which these properties could be verified monolithically.

In summary, we could only verify a larger system using our automated assume-guarantee technique on about 18% of the properties for FLAVERS and on no properties for LTSA. If we had not encountered model building issues, we believe that compositional analysis could verify a larger system size than monolithic verification on at most 40.6% of the properties for FLAVERS and 16.7% of the properties for LTSA. While a 40% success rate may look encouraging, compositional analysis using generalized decompositions did not significantly increase the size of the systems that could be verified. Considering the amount of time that was spent to find the best decomposition at size 2, it is questionable if the benefit of verifying a property on a slightly larger system size is worth the necessary investment of time.

While these results were discouraging, we were interested in determining if there was some way to classify the systems and properties to determine which problems assume-guarantee reasoning would likely produce a significant memory savings. Unfortunately, we could not find such a classification, though we do have some observations. One type of feasibility constraint used by FLAVERS is a *Task Automaton* (TA). Without TAs, analyses in FLAVERS are partially flow-insensitive. Adding a TA for a task makes FLAVERS flow-sensitive with respect to that task. It appears that when more TAs are needed to prove a property, assume-guarantee reasoning based on generalized decompositions is more likely to use more memory than monolithic verification. Of the 14 properties where the number of TAs needed to prove the property increases as the

⁸Recall that by size, as described in section 3, we mean the number of repeated tasks in a system. Thus, increasing the size by two means adding two more of these repeated tasks to a system.

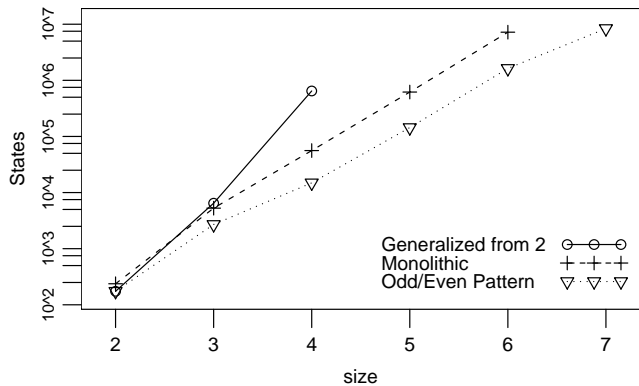


Figure 4: States Explored on the “Always 1 Between 0” Property of Relay with FLAVERS

system size increases, 10 of them are classified as failures in Table 3. Of the 13 properties where only one TA is needed to prove the property regardless of system size, 7 are classified as failures in Table 3, though 3 of these failures are in category 5 where assume-guarantee reasoning based on generalized decompositions does better than monolithic verification but assume-guarantee reasoning is not likely to be of much use since monolithic verification scales well. On the remaining 5 properties, we cannot find a pattern to determine whether or not assume-guarantee reasoning based on generalized decompositions will perform better than monolithic verification. For LTSA, the picture is less clear. On some of the properties where assume-guarantee reasoning for FLAVERS works well, assume-guarantee reasoning for LTSA does not work well. Conversely, there was 1 property where assume-guarantee reasoning for FLAVERS works poorly, but assume-guarantee reasoning for LTSA works well. While these observations may provide some guidance to help determine whether or not assume-guarantee reasoning is likely to save memory, more experimentation is needed before any conclusions can be drawn.

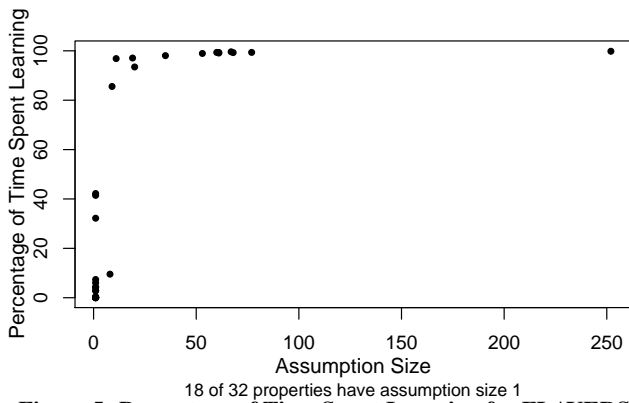
3.4 Are the Generalized Decompositions the Best Decompositions?

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.

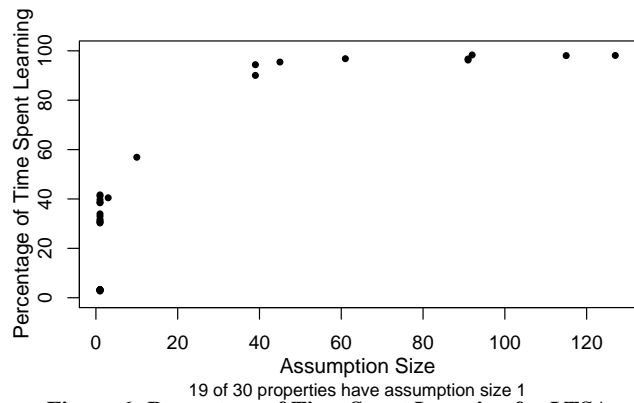
We considered only some of the systems because of the time costs involved. We encountered a number of two-way decompositions where it took more than a month to learn an assumption.⁹ As a result, we imposed an upper bound on the amount of time we spent evaluating a single two-way decomposition to be the maximum of 1 hour and 10 times the amount of time needed to verify that property monolithically.¹⁰ On the largest-sized systems that we completed and for which such a comparison can be made, the generalized decompositions from size 2 are the best decompositions.

⁹The time needed to evaluate the decompositions that took more than a month is counted in the 1.43 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 evaluating just the decompositions that took less than 1 day to evaluate added up to over 10 months of CPU time.

¹⁰On every property where the upper bound on time was reached, we were able to find at least one decomposition that was better than the generalized decomposition.



18 of 32 properties have assumption size 1
Figure 5: Percentage of Time Spent Learning for FLAVERS



19 of 30 properties have assumption size 1
Figure 6: Percentage of Time Spent Learning for LTSA

tions on 31.3% of the properties with FLAVERS and 40.0% of the properties with LTSA.

When we found a decomposition at size n ($n > 2$) that was better than the generalized decomposition for size 2, we generalized the decomposition for size n to systems larger than size n . In all cases, the generalized decomposition for size n is better than the generalized decomposition for size 2. We also tried taking the decompositions for size n and simplifying them so they could be used on systems of size 2, similar to the process described in Appendix A 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 have 3 examples where there are decompositions that can be used to verify a property on a larger system size than both monolithic verification and the generalized decompositions from size 2. One of these examples is the “mutual exclusion” property of the Gas Station system with FLAVERS. Figure 3 compares the number of states explored using two different generalizations (from size 2 and size 4) to the number of states explored using monolithic verification. On this example, the generalized decomposition for size 2 is the best decomposition for size 3. When the system size is 4, however, the generalized decomposition for size 2 is not the best decomposition. We do not know what the best decomposition for size 4 is 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 property on systems with size 6, one size greater than monolithic verification and the generalized decomposition for size 2.

On one of the other examples, the “always 1 between 0” property of the Relay system with FLAVERS, the generalized decompositions for size 2 are worse than monolithic. We were able to find a decomposition that could allow us to verify this property on the system with size 7, one size larger than monolithic verification. However, this decomposition 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, which depends on whether or not the size of the system is odd or even. Figure 4 compares the number of states using the generalized decomposition for size 2, the decompositions based on the odd/even pattern, and the monolithic verification for this property.

On these two examples, we were able to find decompositions that could scale farther than the generalized decomposition for size 2, but at significant expense, since it involved trying all two-way decompositions for larger system sizes; this approach is probably too costly to be useful in practice.

3.5 What is the Cost of Using the L* Algorithm?

In addition to evaluating our use of generalized decompositions, we were also interested in investigating whether or not using the L* algorithm to learn assumptions increased the cost of compositional analysis. To do this, we first determined, for each property, the cost of compositional analysis when the L* algorithm is used to learn an assumption. At the end of each compositional analysis, we saved the assumption that was used to complete the assume-guarantee proof. These assumptions were then used to evaluate the cost of compositional analysis when assumptions are not learned. For each 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 compositional analysis with the L* algorithm. For each property, we compared the time and memory costs for the largest sized system on which that property could be verified using automated assume-guarantee reasoning with the decompositions generalized from size 2.

For a property P , we consider the amount of memory used during compositional analysis 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 compositional analysis 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 properties. On the remaining three properties, on average, compositional analysis without the L* algorithm uses 88.6% of the memory of compositional analysis with the L* algorithm. On one of these properties, compositional analysis was able to scale at least three sizes farther than monolithic verifications. On the other two properties, the memory used by compositional analysis without the L* algorithm was about 10 times more than the memory used by monolithic verification.

For LTSA, the amount of memory used by the two approaches is the same on 26 of the 30 properties. On the remaining four properties, on average, compositional analysis without the L* algorithm uses 65.9% of the memory of compositional analysis with the L* algorithm. On three of these properties, the memory used by compositional analysis without the L* algorithm was about 2 times more than the memory used by monolithic verification. On the fourth property, the amount of memory used by compositional analysis without the L* algorithm was 78.1% of the memory used by monolithic verification.

To summarize, on one of the properties where compositional analysis with the L^* algorithm used more memory than compositional analysis without the L^* algorithm, compositional analysis with the L^* algorithm could scale at least three sizes farther than monolithic verification. This allowed us to increase the size of the system that could be verified from 47 to 50. Since monolithic verification, however, could scale to size 47, a fairly large system size, we do not believe that verifying the system on size 50 is particularly useful. On one of the other properties, compositional analysis without the L^* algorithm used 78.1% of the memory used by monolithic verification, so it is unlikely that compositional analysis without the L^* algorithm would be able to scale significantly farther than monolithic verification. On the remaining properties, compositional analysis without the L^* algorithm used more memory than monolithic verification. Unless a better assumption and decomposition could be found for these properties, it is unlikely that compositional analysis would be able to scale farther than monolithic verification. While such assumptions and decompositions may exist, we do not know of a good way to find them and do not believe that it will be easy for analysts to find them without some form of automated support.

To determine the time cost for using the L^* algorithm to learn an assumption, we looked at the amount of time that was needed to learn an assumption to verify each property. Figures 5 and 6 show the percentage of time that was 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 of the learned assumption is larger than 10 states, however, in most cases over 90% of the verification time is spent learning the assumptions, a substantial overhead.

Because of this time and memory 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 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, Groce et al. developed a technique for initializing some of the L^* algorithm’s data structures when given an automaton that recognizes a language close to the one being learned [22]. By doing this, they reduced the amount of time needed to run Angluin’s version of the L^* algorithm. To determine if this technique would reduce the cost of using learning, for each property, we used the L^* algorithm to learn an assumption capable of completing an assume-guarantee proof. This assumption was then used to initialize some of the data structures of Angluin’s version of the L^* algorithm. Performing this initialization reduced the number of queries made by the L^* algorithm (and consequently the running time) when compared to not initializing these data structures. Initializing these data structures in Angluin’s version of the L^* algorithm, however, 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.

Using the L^* algorithm to learn an assumption can increase both the time and memory cost needed to complete an assume-guarantee proof, compared to the cost of completing a proof using a supplied

assumption. Some of the learned assumptions are very large: in fact, one has over 250 states. For such systems, analysts cannot be expected to develop these assumptions manually and we do not believe that small assumptions exist that can be used to complete the assume-guarantee proof. Thus, some automated support is needed to make assume-guarantee reasoning practical on these systems.

3.6 Threats to Validity

While 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; we examined over 43,000 two-way decompositions and used over 1.43 years of CPU time.

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. [1] for NuSMV [9] in which they found some examples where assume-guarantee reasoning could verify a larger system than monolithic verification and other examples where assume-guarantee reasoning used more memory than monolithic verification.

We looked only at one assumption generation technique, which influenced the assumptions that we used in completing the assume-guarantee proofs. Although other assumptions could be used 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., [1, 5, 6, 21, 27]) 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 do not expect that using these other techniques will produce better results. While techniques based on assume-guarantee rules that allow for more than two-way decompositions (e.g., [14, 24, 26]) might perform better with respect to memory, there has not yet been enough empirical evaluation of these techniques to draw any such conclusions.

Additionally, we looked only at a small number of systems that were mostly based on a client-server architecture and where scaling was achieved by replicating the number of clients. This allowed us to easily increase the size of the system to look at the effects of scaling on assume-guarantee reasoning. Looking at just one kind of scaling, however, is a threat to the validity of our results. Alur et al., however, looked at systems with different architectures and had results that were similar to ours [1].

4. RELATED WORK

Our work uses the automated assume-guarantee approach of Cobleigh et al., which uses the L^* algorithm to learn assumptions to complete assume-guarantee proofs [11]. Several other assume-guarantee reasoning approaches based on the L^* algorithm have been proposed. The work of Barringer et al. extends the approach of [11] to use symmetric assume-guarantee rules [5]. Chaki et al. developed an algorithm based on the L^* algorithm for learning tree automata for checking simulation conformance [6]. Alur et al. [1] adapted the L^* algorithm for use with NuSMV. They found some examples that could be verified using assume-guarantee reasoning but could not be verified monolithically. Some of these examples were scalable systems and, on these systems, they were able to increase the size of the system that could be verified by 1 or 2. They did not, however, determine if assume-guarantee reasoning could scale farther than this, but, based on their data, it seems unlikely.

Alur et al. also reported on one example where assume-guarantee reasoning used more time and memory than monolithic verification. The work of Giannakopoulou et al. also computes assumptions, but requires exploring the entire state space of S_1 [21]. Since this may not be necessary when the L^* algorithm is used, we believe the approach using the L^* algorithm is more scalable.

Henzinger et al. have presented a framework for thread-modular abstraction refinement, in which assumptions and guarantees are both refined in an iterative fashion [24]. This framework applies to programs that communicate through shared variables, and, unlike our approach, is not guaranteed to terminate. The work of Flanagan and Qadeer also focuses on a shared-memory communication model [17], but does not address notions of abstractions as is done in [24]. Jeffords and Heitmeyer use an invariant generation tool to generate invariants for components that can be used to complete an assume-guarantee proof [27]. While their proof rules are sound and complete, their invariant generation algorithm is not guaranteed to produce invariants that will complete an assume-guarantee proof even if such invariants exist.

Another compositional analysis approach that has been advocated is Compositional Reachability Analysis (CRA) (e.g., [19, 39]). 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., [8]) has been shown to reduce the cost of verification. Still, CRA is hampered by the state explosion problem. Although constraints, both manually supplied and automatically derived, can help reduce the cost of CRA [8], determining how to apply CRA to effectively reduce the cost of verification still remains a difficult problem.

5. CONCLUSIONS

In this work, we explored the question of whether or not assume-guarantee reasoning provides an advantage over monolithic verification. Unfortunately, the results of our experiments are not very encouraging. The vast majority of decompositions explored more states than monolithic verification. While this is not surprising, it is worth noting. The process of examining all two-way decompositions is too costly to be useful in practice and we do not have a good way to predict whether or not a given decomposition will save memory over monolithic verification. Thus, even in those cases when assume-guarantee reasoning can save memory over monolithic verification, it is unclear how analysts will be able to find those decompositions without some guidance on how to decompose the system being analyzed.

If we restrict our attention to just the best decomposition at the smallest size for each example, then in only about half of the cases we examined did the assume-guarantee reasoning technique we used explore fewer states than monolithic verification. For the cases where there is a memory savings with FLAVERS, assume-guarantee reasoning used, on average, 48.4% of the memory used by monolithic verification. With LTSA, this percentage was 33.6%. We were most interested in determining if this memory savings would be substantial enough to allow us to verify properties on larger systems than could 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 could save memory over monolithic verification, there were very few cases in which this savings was sufficient to allow verification of a larger system. Fur-

thermore, for the cases where assume-guarantee reasoning could verify a larger system than monolithic verification, it could not significantly increase the size of the system that could be verified.

Of course, there are decompositions other than the generalized ones that we could have tried on larger systems. In fact, we know that there are some decompositions for some examples that can be used to verify larger systems than what the generalized decompositions can be used to verify. We were unable to find such decompositions using our intuition and did not observe any pattern that could be used to select a good decomposition for a given system.

Although these results are preliminary, they raise doubts about the usefulness of assume-guarantee reasoning as an effective compositional analysis technique. In our experiments, we found that assume-guarantee reasoning only rarely allowed us to verify larger systems than could be verified monolithically. While automated assume-guarantee reasoning techniques can make compositional analysis easier to use, determining how to apply these techniques most effectively is still difficult, can be expensive, and may not significantly increase the sizes of the systems that can be verified. Clearly additional experiments should be done using other automated learning techniques, other verification systems, and other compositional approaches. The negative results presented here, however, increase the need for strong empirical justification of new compositional approaches.

6. REFERENCES

- [1] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In Etessami and Rajamani [16], pages 548–562.
- [2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.
- [3] G. S. Avrunin, J. C. Corbett, and M. B. Dwyer. Benchmarking finite-state verifiers. *Int. J. on Soft. Tools for Tech. Transfer*, 2(4):317–320, 2000.
- [4] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Păsăreanu, and S. F. Siegel. Comparing finite-state verification techniques for concurrent software. TR 99-69, U. of Massachusetts, Dept. of Comp. Sci., Nov. 1999.
- [5] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proc. of the Second Workshop on Spec. and Verification of Component-Based Systems*, pages 14–21, Sept. 2003.
- [6] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In Etessami and Rajamani [16], pages 534–547.
- [7] R. Chatley, S. Eisenbach, and J. Magee. MagicBeans: a platform for deploying plugin components. In W. Emmerich and A. L. Wolf, editors, *Proc. of the Second Int. Working Conf. on Component Development*, volume 3083 of *LNCS*, pages 97–112, May 2004.
- [8] S.-C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Trans. on Soft. Eng. and Methodology*, 5(4):334–377, Oct. 1996.
- [9] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An open source tool for symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Proc. of the Fourteenth Int. Conf. on Computer-Aided Verification*, volume 2404 of *LNCS*, pages 359–364, July 2002.
- [10] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke. Breaking up is hard to do: An investigation of decomposition for assume-guarantee reasoning. TR UM-CS-2004-023, U. of Massachusetts, Dept. of Comp. Sci., Apr. 2004.
- [11] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In H. Garavel and J. Hatcliff, editors, *Proc. of the Ninth Int. Conf. on Tools and Alg. for the Construction and Analysis of Sys.*, volume 2619 of *LNCS*, pages 331–346, Apr. 2003.

- [12] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97–123, Jan. 1995.
- [13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd Int. Conf. on Soft. Eng.*, pages 439–448, June 2000.
- [14] C. de la Riva and J. Tuya. Modular model checking of software specifications with simultaneous environment generation. In F. Wang, editor, *Proc. of the Second Int. Conf. on Automated Tech. for Verification and Analysis*, volume 3299 of *LNCS*, pages 369–383, Oct.-Nov. 2004.
- [15] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent software systems. *ACM Trans. on Soft. Eng. and Methodology*, 13(4):359–430, Oct. 2004.
- [16] K. Etesami and S. K. Rajamani, editors. *Proc. of the Seventeenth Int. Conf. on Computer-Aided Verification*, volume 3576 of *LNCS*, July 2005.
- [17] C. Flanagan and S. Qadeer. Thread-modular model checking. In T. Ball and S. K. Rajamani, editors, *Proc. of the Tenth SPIN Workshop*, volume 2648 of *LNCS*, pages 213–224, May 2003.
- [18] C. Fournet, T. Hoare, S. K. Rajamani, and J. Rehof. Stuck-free conformance. In R. Alur and D. Peled, editors, *Proc. of the Sixteenth Int. Conf. on Computer-Aided Verification*, volume 3114 of *LNCS*, pages 242–254, July 2004.
- [19] D. Giannakopoulou, J. Kramer, and S.-C. Cheung. Behaviour analysis of distributed systems using the Tracta approach. *Automated Soft. Eng.*, 6(1):7–35, Jan. 1999.
- [20] D. Giannakopoulou and C. S. Păsăreanu. Learning-based assume-guarantee verification. In P. Godefroid, editor, *Proc. of the Twelfth SPIN Workshop*, volume 3639 of *LNCS*, pages 282–287, Aug. 2005.
- [21] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proc. of the Seventeenth IEEE Int. Conf. on Automated Soft. Eng.*, pages 3–12, Sept. 2002.
- [22] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proc. of the Eighth Int. Conf. on Tools and Alg. for the Construction and Analysis of Sys.*, number 2280 in *LNCS*, pages 357–370. Springer-Verlag, Apr. 2002.
- [23] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, Mar. 1985.
- [24] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In W. A. Hunt, Jr. and F. Somenzi, editors, *Proc. of the Fifteenth Int. Conf. on Computer-Aided Verification*, volume 2725 of *LNCS*, pages 262–274, July 2003.
- [25] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In A. J. Hu and M. Y. Vardi, editors, *Proc. of the Tenth Int. Conf. on Computer-Aided Verification*, volume 1427 of *LNCS*, pages 440–451, June–July 1998.
- [26] P. Inverardi, A. L. Wolf, and D. Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Trans. on Soft. Eng. and Methodology*, 9(3):239–272, July 2000.
- [27] R. D. Jeffords and C. L. Heitmeyer. A strategy for efficiently verifying requirements. In *Proc. of the Ninth European Soft. Eng. Conf. held jointly with the Eleventh ACM SIGSOFT Symp. on the Foundations of Soft. Eng.*, pages 28–37, Sept. 2003.
- [28] C. B. Jones. Specification and design of (parallel) programs. In R. Mason, editor, *Information Processing 83: Proc. of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.
- [29] R. K. Keller, M. Cameron, R. N. Taylor, and D. B. Troup. User interface development and software environments: The Chiron-1 system. In *Proc. of the Thirteenth Int. Conf. on Soft. Eng.*, pages 208–218, May 1991.
- [30] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
- [31] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [32] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proc. of the Sixth ACM SIGSOFT Symp. on the Foundations of Soft. Eng.*, pages 24–34, Nov. 1998.
- [33] S. S. Patil. Limitations and capabilities of Dijkstra’s semaphore primitives for coordination among processes. Computational Structures Group Memo 57, Project MAC, Feb. 1971.
- [34] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.
- [35] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI*, pages 123–144. Springer-Verlag, Oct. 1984.
- [36] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, Apr. 1993.
- [37] S. F. Siegel and G. S. Avrunin. Improving the precision of INCA by eliminating solutions with spurious cycles. *IEEE Trans. on Soft. Eng.*, 28(2):115–128, Feb. 2002.
- [38] R. N. Taylor, F. C. Belz, L. A. Clarke, L. J. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young. Foundations for the Arcadia environment architecture. In *Proc. of the ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on Practical Soft. Development Environments*, pages 1–13, Nov. 1988.
- [39] W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proc. of the 1991 Symp. on Testing, Analysis, and Verification*, pages 49–59, Oct. 1991.

```

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;

```

Figure 7: Elevator system in Ada

APPENDIX

A. GENERATING DECOMPOSITIONS FOR LARGER SYSTEM SIZES

The process we used for generating decompositions for larger sizes given the best decomposition for size 2 is as follows:

- For each non-repeatable 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 . Otherwise, 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 at the property. Often, the property treated one of the repeatable tasks in a different way than all the other repeatable tasks.
 - + If one of the repeatable tasks is treated in a different way in the property, 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 one of the repeatable tasks is not treated in a different way, then, in the properties in our case study, all of the repeatable tasks are treated the same way.
 - 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_1 in the best decomposition at size 2. Otherwise, put this repeatable task into S_1 .

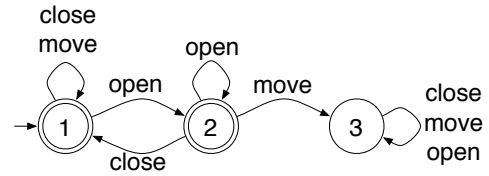


Figure 8: Example property for FLAVERS

B. THE TEACHER FOR FLAVERS

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.

B.1 Detailed Description of FLAVERS

Consider the example shown in Figure 7, 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 uses a race condition so that the variable of 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.

The properties that FLAVERS verifies 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 is translated into a *Finite-State Automaton* (FSA). 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 8, 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 non-accepting state, represents a violation of the property, since

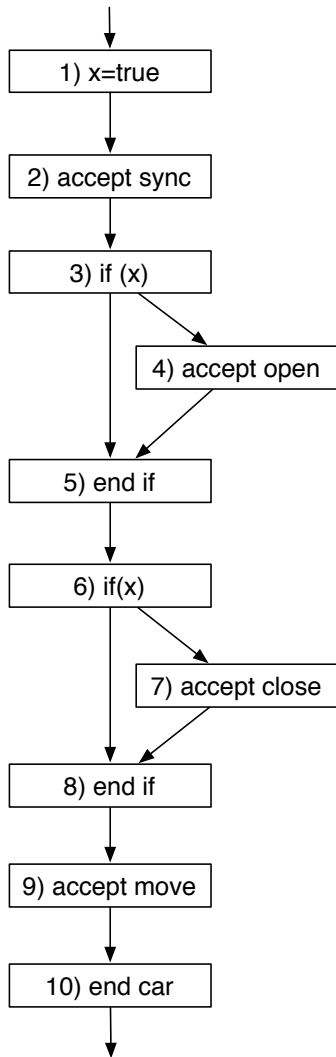


Figure 9: CFG for task car

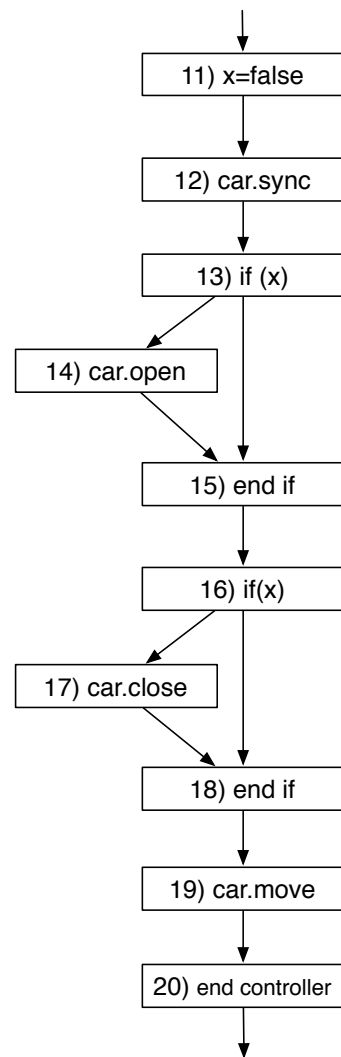


Figure 10: CFG for task controller

the only way to enter it is 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 *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. The CFGs for the car and controller tasks are shown in Figures 9 and 10, 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 8 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. This refinement is safe so long as there is a weak bisimulation relationship [31] between each original CFG and its corresponding refined CFG. Figures 11 and 12 show the CFGs for the car and controller tasks refined with respect to the property shown in Figure 8. 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.

Since a CFG corresponds to the 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 concurrent systems. 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 11 and 12 is shown in Figure 13. 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 [32] and are shown as dashed edges in Figure 13.

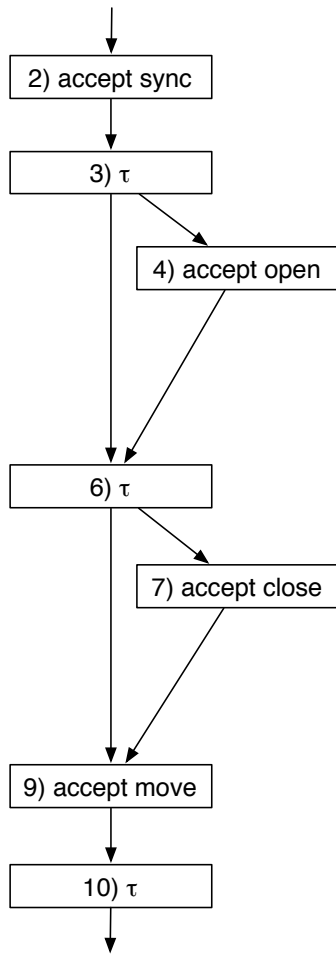


Figure 11: Refined CFG for task car

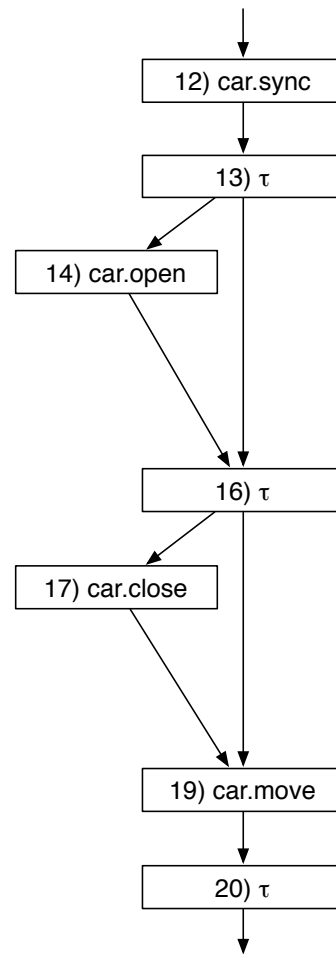


Figure 12: Refined CFG for task controller

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.

A TFG is an over-approximation of the sequences of events that can occur when executing a system. 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, there usually are paths in the TFG that do not correspond to any actual execution of the system. Full details of TFG construction and the proof of TFG conservativeness are given in [15].

FLAVERS uses an efficient state-propagation algorithm 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 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. *Infeasible paths* do not correspond to any possible execution of the system but are an artifact of the imprecision of the model.

FLAVERS would report that the property shown in Figure 8

does not hold on the TFG shown in Figure 13 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 $\langle \text{open}, \text{move} \rangle$. 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 infeasible.

This infeasible path was introduced because all information related to the variable x was removed during CFG refinement. In order to improve the precision of the model and remove some infeasible paths from consideration, the analyst can introduce *feasibility constraints*, also represented as FSAs. An analyst might need to iteratively add feasibility constraints and observe the analysis results several times before determining whether or not a property holds. Feasibility constraints give analysts some control over the analysis process by letting them determine exactly what parts of a system need to be modeled in order to prove a property.

One type of feasibility constraint is a *Variable Automaton (VA)*, which can be used to track a small number of values for a variable. In this example, the value of the variable x is important to the analysis, so a VA can be introduced to keep track of its value. In the VA

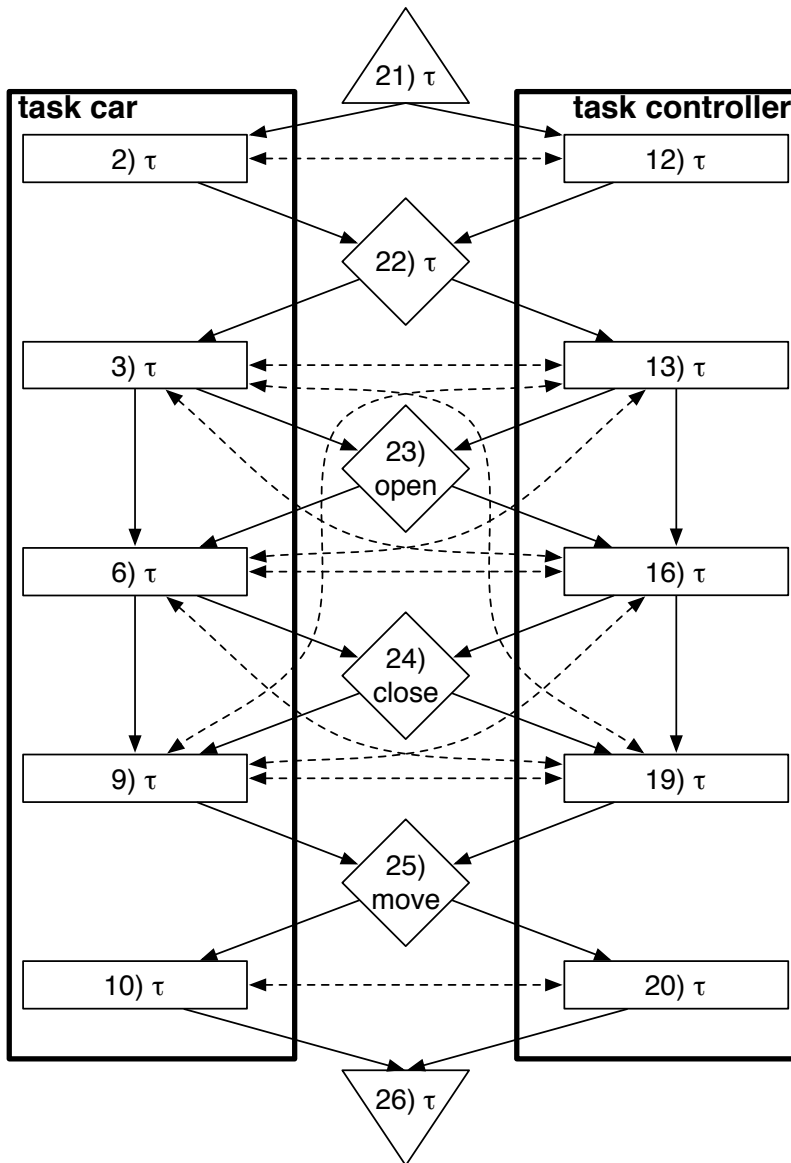


Figure 13: TFG for the elevator system

for x , shown in Figure 14, 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 non-accepting 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 $x==\text{false}$ occurs), then the violation state would be entered.

To make use of this VA, the TFG from Figure 13 needs to be modified so it has nodes with events corresponding to operations on x , as shown in Figure 15. In this TFG,¹¹ each node corresponding to a branch (nodes 3, 6, 13, and 16 from Figure 13) has been split into two nodes, one for the true branch (the nodes labeled $x==\text{true}$) and one for the false branch (the nodes labeled $x==\text{false}$).

¹¹In our implementation, nodes 2 and 12 would be refined away, but they have been left in Figure 15 for clarity of the presentation.

The counterexample path reported previously corresponds to the path $21 \rightarrow 1 \rightarrow 2 \rightarrow 22 \rightarrow 3a \rightarrow 23 \rightarrow 6b \rightarrow 9 \rightarrow 25 \rightarrow 10 \rightarrow 26$ in the modified TFG. The path corresponds to the event sequence $\langle x=\text{true}, x==\text{true}, \text{open}, x==\text{false}, \text{move} \rangle$. 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 is infeasible. With just the VA for x , FLAVERS would report that the property shown in Figure 8 holds and that the elevator’s car cannot move while its doors are open.

It is important to note that there are still infeasible paths in this model. For example, any path that starts $21 \rightarrow 11 \rightarrow 2$ is infeasible because it visits node 2 without having first visited node 1. A *task automaton* can be used to remove such infeasible paths from consideration. A task automaton enforces the flow of control within a single task of a system. Unlike VAs, which have transitions based

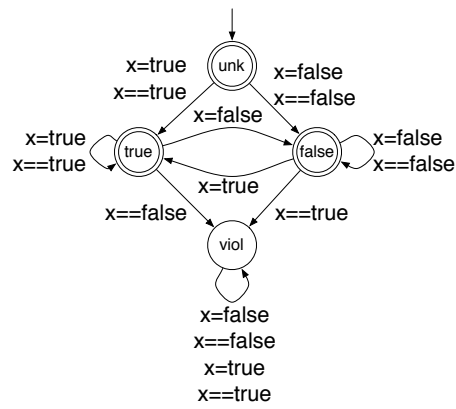


Figure 14: Variable automaton for x

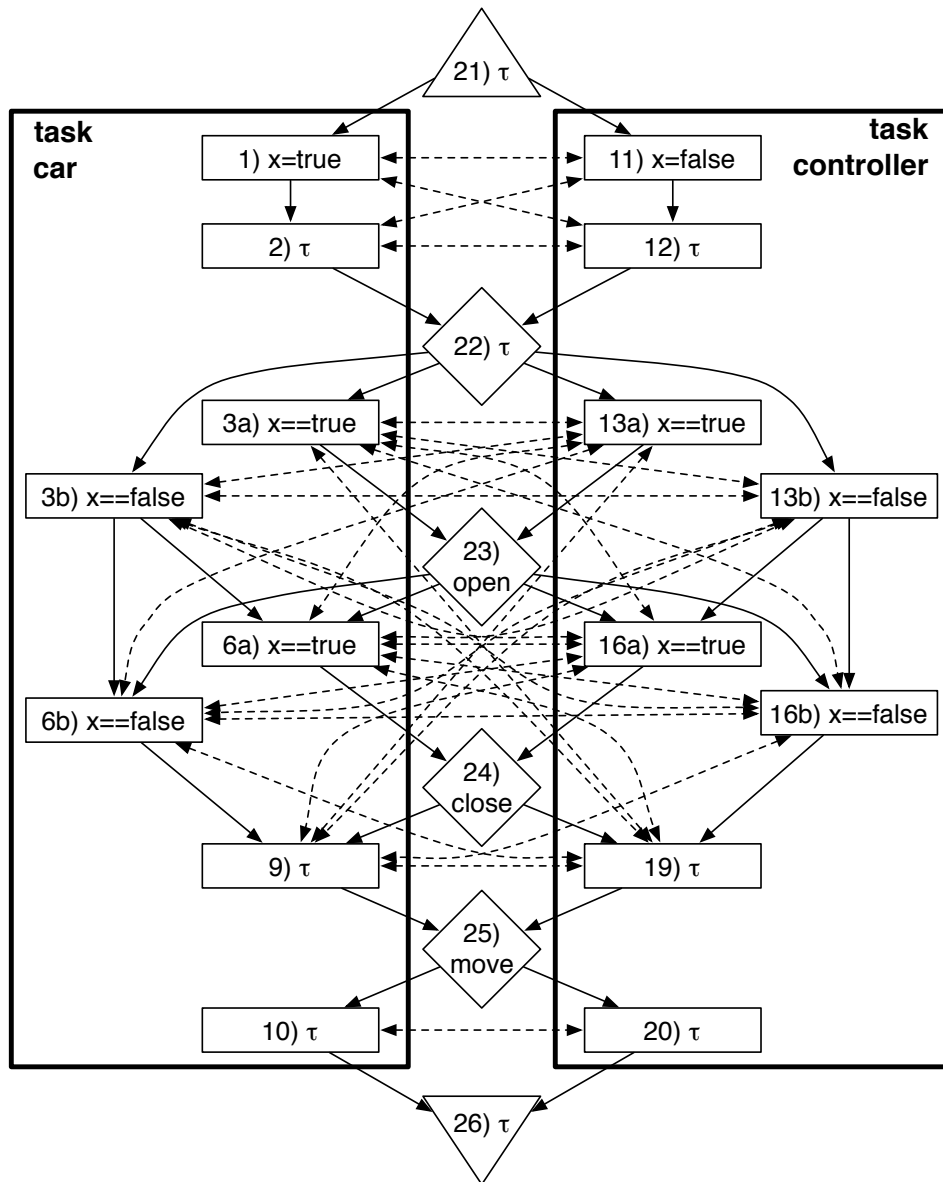


Figure 15: TFG with events for x

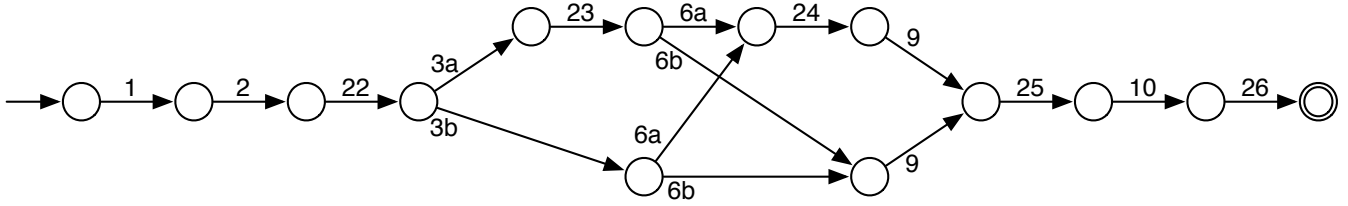


Figure 16: TA for task car

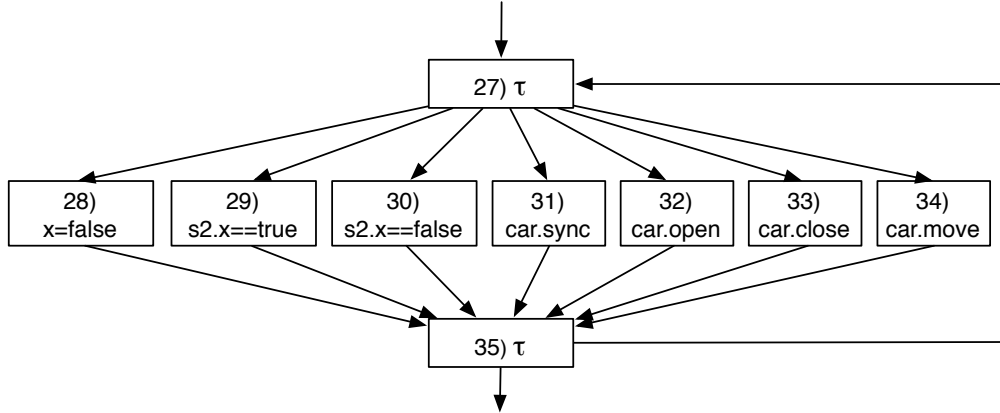


Figure 17: Environment for S_1

on the annotations on the TFG nodes, TAs have transitions based on the IDs of the TFG nodes. The TA for the car task is shown in Figure 16. Unlike previous FSAs in which have been total,¹² we omit transitions from this FSA that cannot ever lead to an accepting state for simplicity. This TA would prevent paths starting $21 \rightarrow 11 \rightarrow 2$ from being considered during verification since, from the initial state of the TA, if any node from the car task other than node 1 is visited, the TA will not accept that sequence. Since the first node visited from the car task on these paths is node 2, the TA would not accept these paths.

B.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. 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 8. Suppose that $S_1 = \text{car}$ and $S_2 = \text{controller}$.

B.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. If the TFG for S_1 were built just from the CFG for the car task, then this TFG would not have any entry calls made to the rendezvous it accepts. Thus, to model S_1 in the context of the whole system, an environment needs to be constructed to represent interactions be-

tween S_1 and S_2 .

The environment for S_1 needs to have accept statements¹³ from S_2 that are called by S_1 and entry calls made by S_2 to accept statements in S_1 . In this example, these would be the entry calls `close_doors`, `move_car`, `open_doors`, and `sync`.

Additionally, the environment for S_1 needs to contain events from S_2 that can affect the property or constraints. To determine these events, each property and constraint automaton that contains events in both S_1 and S_2 is examined. For these automata, events that occur in S_2 need to be added to the environment of S_1 . In this example, 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 . Now, the events `x==true` and `x==false` also occur in S_1 . When checking $\langle A \rangle S_1 \langle P \rangle$, for example, we want the assumption A to only constrain the behavior of the environment of S_1 , not the behavior of S_1 . Thus, we need to relabel the property and constraints that have events in both S_1 and S_2 so that an event that is common to S_1 and S_2 can be distinguished. We prefix common events in S_1 with “s1” and events in S_2 with “s2”. The events on the nodes of the CFGs for tasks in S_1 and S_2 need to be similarly relabeled.

Finally, the environment needs to be able to perform, in any order, zero or more accepts, entry calls, and events that can affect the property of constraints. All of the information needed to construct the environment can be gathered from the CFGs and constraints, so the environment can be automatically constructed. Figure 17 shows the CFG for the environment for S_1 . Notice that nodes 29

¹²A total FSA is one which has a transition from every state on every event in its alphabet.

¹³Ada also has accept statements with bodies. To support this construct, the single rendezvous with a body is converted into two rendezvous: one that occurs before the body and one that occurs after the body.

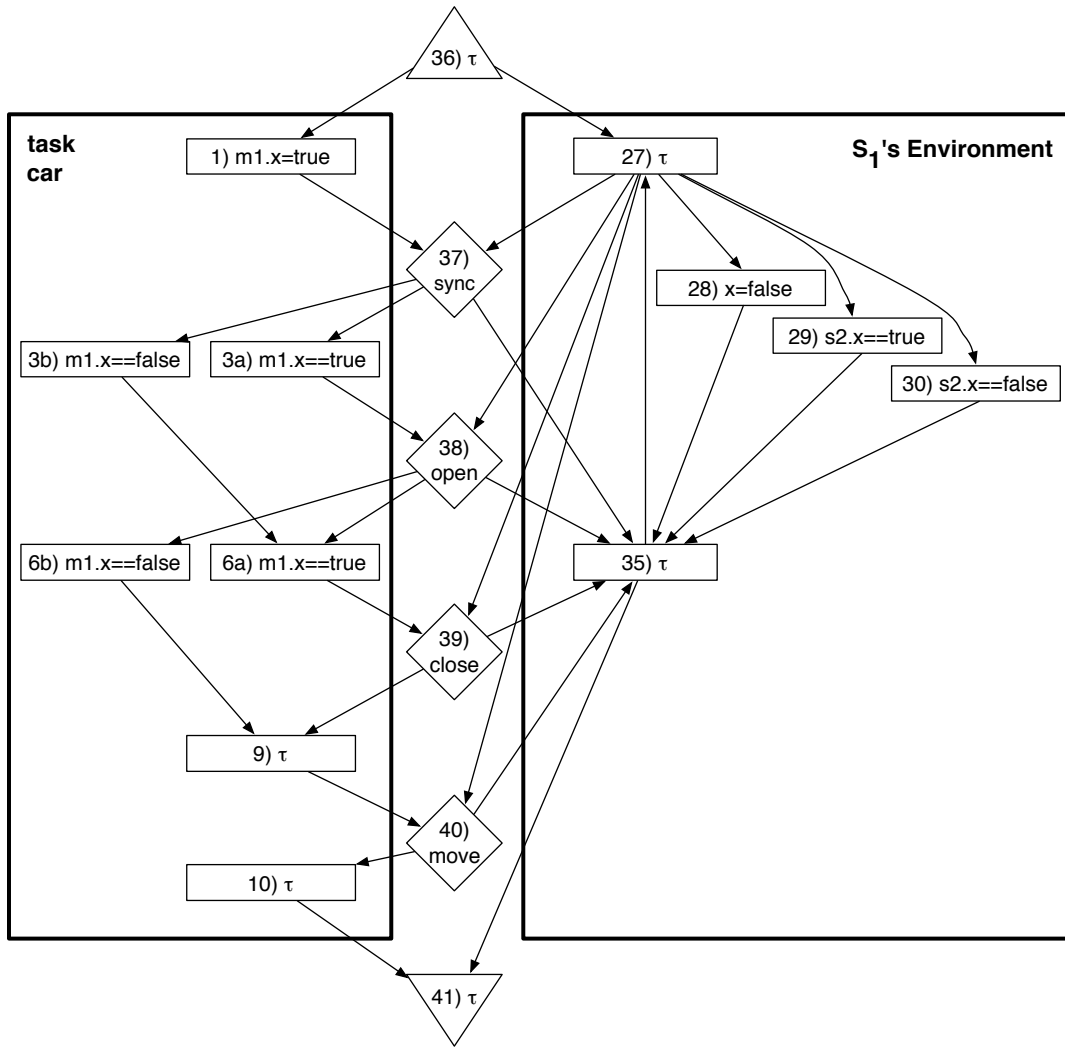


Figure 18: TFG for S_1 , without MIP edges

and 30, which are labeled with events common to S_1 and S_2 have been prefixed with “s2”. 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 18 shows the TFG for S_1 in the elevator example, with MIP edges removed for clarity.

B.2.2 The alphabet of the assumption

The L^* algorithm learns an FSA over an alphabet Σ . In order to use the L^* algorithm for assume-guarantee reasoning, it needs to be given Σ . For FLAVERS, the alphabet consists of the labels on all rendezvous¹⁴ 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, $\Sigma_A = \{\text{close}, \text{s2.x==false}, \text{s2.x==true}, \text{x=false}, \text{move}, \text{open}, \text{sync}\}$.

B.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

¹⁴This includes rendezvous that are not mentioned in the property or constraints. This is why node 37 in Figure 18 is labeled with sync instead of τ .

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 . Of the constraints provided by the analyst, this verification uses the TAs for the tasks in S_1 and the VAs that contain events in S_1 . The CFGs, VAs, and the property P are re-labeled 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 look at paths through the TFG that correspond to the event sequence specified by the query. For example, if the query were $\langle \text{sync}, \text{s2.x==false}, \text{open}, \text{move} \rangle$, then the constraint shown in Figure 19 would be used. State-propagation is then applied to check the property; if the property is violated then false is returned to the

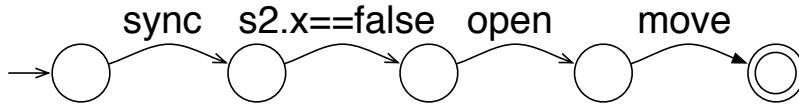


Figure 19: Constraint for the query $\langle \text{sync}, \text{s2.x}==\text{false}, \text{open}, \text{move} \rangle$

L* Algorithm, otherwise true is returned.

B.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 and, the constraints used consist of the TAs for tasks in S_1 and VAs 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 $\langle \text{true} \rangle S_2 \langle A \rangle$ 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 . Rather than treat A as a constraint as was done in checking Premise 1, A is treated as a property. The constraints used consist of the TAs for tasks in S_2 and VAs 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.