

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

Jamieson M. Cobleigh
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
jcobleig@cs.umass.edu

George S. Avrunin
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
avrunin@cs.umass.edu

Lori A. Clarke
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
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. We were interested in determining if assume-guarantee reasoning could provide an advantage over monolithic verification. Using recent advances in assume-guarantee reasoning that automatically generate assumptions, we undertook a study that considered *all* two-way decompositions for a set of systems and properties, using two different verifiers. By increasing the number of repeated tasks for a system, we evaluated the decompositions as the systems were scaled. In very few cases were we able to show that assume-guarantee reasoning could verify properties on larger systems than monolithic verification could. Additionally, assume-guarantee reasoning could only verify these properties on systems a few sizes larger than monolithic verification. This negative result, although preliminary, raises doubts about the usefulness of assume-guarantee reasoning as an effective technique.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*model checking*

General Terms

Verification, Experimentation

*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.

WOODSTOCK '97 El Paso, Texas USA

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

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 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 [20, 25] 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:

$$\frac{\begin{array}{l} \text{(Premise 1)} \quad \langle A \rangle S_1 \langle P \rangle \\ \text{(Premise 2)} \quad \langle \text{true} \rangle S_2 \langle A \rangle \end{array}}{\langle \text{true} \rangle S_1 \parallel S_2 \langle P \rangle}$$

By checking the 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, for example, that 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 the difficulty of doing compositional reasoning, it has not been practical to undertake an empirical evaluation of this approach, although several case studies have been reported [5, 9, 12, 15, 18].

There has been recent work on automatically computing assumptions for compositional analysis [4, 9, 15, 18] that has eliminated one of the obstacles to empirical evaluation. By automatically creating the assumptions, it is now possible to examine a large number of decompositions. Using the algorithm in [9], 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 ex-

pected from assume-guarantee reasoning. For this study, we implemented this automated assume-guarantee reasoning algorithm for two verifiers, FLAVERS [11] and LTSA [22]. We began by looking at several examples using FLAVERS, but the results of these experiments were not as promising as the results seen in [9], 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. We considered all of these two-way decompositions to find the best decomposition, that is, the decomposition that explored the fewest number of states. To determine if assume-guarantee reasoning could be used to verify these properties on larger system sizes than could be verified monolithically, we generalized the best decompositions to make them applicable to larger system sizes. We needed to use generalized assumptions because checking all two-way decompositions to find the best decomposition is infeasible for larger system sizes. To evaluate the generalized decompositions, we looked at two-way decompositions of some larger system sizes. We were not able to find the best decomposition in all cases because of the time cost involved. Still, 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 reasoning 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 (**F**low Analysis for **V**erification of **S**ystems) is a finite-state verifier that can prove user-specified properties of sequential and concurrent systems [11]. 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 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. *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 $\mathcal{O}(N^2 \cdot |Q|)$, 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 [11] and that the performance of FLAVERS is good when compared to other finite-state verifiers [2, 3].

2.2 LTSA

LTSA (**L**abeled **T**ransition **S**ystems **A**nalyzer) is a finite-state verifier that can prove user-specified properties of sequential and concurrent systems [22]. 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 for 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 [13]. 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 [1] and later improved by Rivest and Schapire [26]. 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 by building an observation table through its interactions with a *minimally adequate teacher*, henceforth referred to as a *teacher*. The teacher needs to answer two types of questions, queries and conjectures. A *query* consists of a string in Σ^* and the teacher returns *true* if the string is in the regular language being learned and *false* otherwise. A *conjecture* consists of an FSA that the L* algorithm believes will recognize the language being learned. The teacher returns *true* if the FSA 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.

We use the L* algorithm to learn an assumption to verify a property P with assume-guarantee reasoning as described in [9]. In this approach, the system under analysis needs to be divided into two subsystems, S_1 and S_2 . Then, a teacher that is capable of answering the queries and conjectures made by the L* algorithm must be provided. At a high level, this teacher is the same for both FLAVERS and LTSA, however, differences in the models used by FLAVERS and LTSA necessitated differences in the implementation of their teachers. Space does not permit us to describe the specific teachers for the two verifiers, which are described in [8] for FLAVERS and [9] for LTSA.

A query posed by the L* algorithm consists of a sequence of events from Σ^* . The teacher must answer true if this sequence is in the language being learned and false otherwise. To answer a query, the model of S_1 is examined to determine if the given sequence results in a violation of the property P . 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.

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 a candidate assumption to be used to complete an assume-guarantee proof, 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.

First, a query is made 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.

It was shown in [9] that this approach to assume-guarantee reasoning will terminate. Using Rivest and Schapire’s L* algorithm, $l - 1$ conjectures and $\mathcal{O}(kl^2 + l \log m)$ 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.

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 [9] 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 [21], both the single and the multiple dispatcher versions as described in [3], the Gas Station problem [17], Peterson’s mutual exclusion problem [24], the Relay problem [27], and the Smokers problem [23].

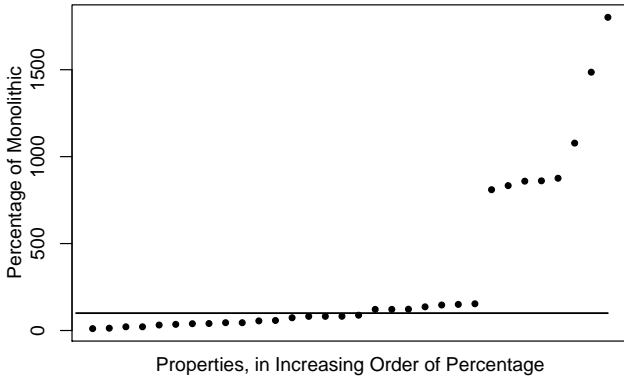
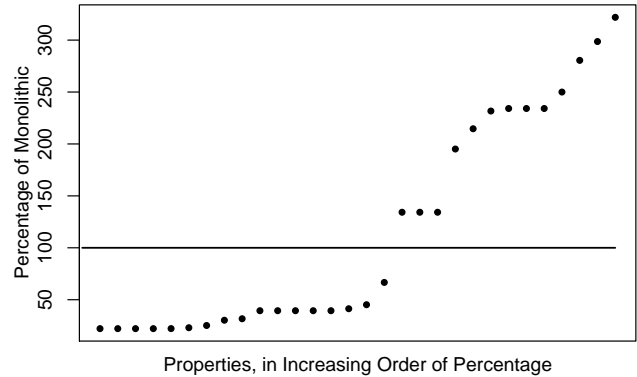
Both FLAVERS and LTSA prove that a property holds by exploring all of the reachable states in an abstracted model of a system. On properties that do not hold, these two tools stop as soon as a property violation is found. As a result, their performance is more variable. While using only properties that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

We implemented this technique in two verifiers, FLAVERS and LTSA. Since FLAVERS uses constraints to control the amount of precision in a verification, we used a minimal set of constraints when verifying properties monolithically and compositionally¹. We did not use the most recent version of LTSA, which is based on plugins [6], 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 [14], 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

¹A 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. Some of these sets were found for [11], and the remainder were found using the process described in that paper. While the worst-case complexity of FLAVERS increases with each constraint that is added, sometimes adding more constraints can improve the performance of FLAVERS.

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**

the parts of the model that do not change between different queries and conjectures.

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

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, that is, a partitioning of the tasks of that system into S_1 and S_2 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 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. For each property, we examined *all* two-way decompositions of that system at size 2 to find the best decomposition with respect to memory.

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 an-

swered. We say one decomposition is better than another decomposition if the maximum number of states explored when the teacher answers a query or conjecture when using the first decomposition is smaller than the maximum number of states explored when the teacher answers a query or conjecture when 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.

Table 1 lists, for FLAVERS and LTSA, the number of properties for each system², the number of two-way decompositions³ 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

²There 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.

³Note that the number of two-way decompositions examined for each property are always two less 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.

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	18	56.3%	18	56.3%
LTSA	30	17	56.7%	5	16.7%

this line represent properties on which the best decomposition is better than monolithic verification.

The best decomposition is better than monolithic verification on 17 of 32 properties with FLAVERS. 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.

The best decomposition is better than monolithic verification on 17 of 30 properties with LTSA. There is some overlap between the 17 properties on which the best decomposition is better than monolithic verification with FLAVERS and LTSA, but these sets of 17 properties are different for the two verifiers. For the 17 properties where the best decomposition is better than monolithic verification with LTSA, 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 each property are not better than monolithic verification. Thus, randomly selecting decompositions would likely not yield a decomposition better than monolithic verification. Before we looked at all two-way decompositions for the systems of size 2, we also tried selecting decompositions manually for each property based on our understanding of the systems and properties. In many cases, we were able to find decompositions that used less memory than monolithic verification, when such decompositions existed. In very few cases, however, did we select the best decomposition.

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 18 properties on which the best decomposition at size 2 is better than monolithic verification and 18 properties on which the generalized decomposition is better than monolithic verification, these are not exactly the same 18 properties. 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 is also 1 property on which the best decomposition at size 2 is worse than monolithic verification, but the generalized decomposition is better than monolithic verification.

With LTSA, for most of the properties on which the best decomposition at size 2 is better than monolithic verification, the generalized decomposition is not better than monolithic verification.

In summary, with FLAVERS the best decomposition at size 2 is better than monolithic verification 56.3% of the time and the generalized decomposition is better than monolithic verification 56.3% of the time. However, with LTSA the best decomposition at size 2 is better than monolithic verification 56.7% of the time and the generalized decomposition is better than monolithic verification only 16.7% of the time. Thus, while the automated assume-guarantee reasoning technique we used was frequently not useful for saving memory on larger sized systems, there were still some systems on which it could save memory. This memory savings, however, is probably not much use 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?

To see whether the automated assume-guarantee reasoning technique could verify properties of larger systems than monolithic ver-

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, but generalized unlikely to be of much use	4	12.5%	0	0.0%
	Subtotal	21	59.4%	25	83.3%
Total		32	100.0%	30	100.0%

ification, we used generalized decompositions based on the best decomposition for size 2, as described in Appendix A. We tried to determine, for each property, whether or not compositional analysis using the generalized decompositions 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 FLAVERS⁴ [28] was unable to generate models for the 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. 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 use⁵.

Table 3 shows the number of properties in each category for FLAVERS and LTSA. We consider using generalized decompo-

sitions 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 40.6% for FLAVERS and 16.7% 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: 18.8% for FLAVERS and 0.0% 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 analysis could verify the property on a system one size larger, but not two sizes larger, than monolithic verification could. 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 18.8% 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 40.6% 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

⁴This toolkit is very old and not easily modifiable. We are working on building models for FLAVERS from Java programs using Bandera [10] and, thus, expect to remove some of the limitations of our old language processing toolkit.

⁵The 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.

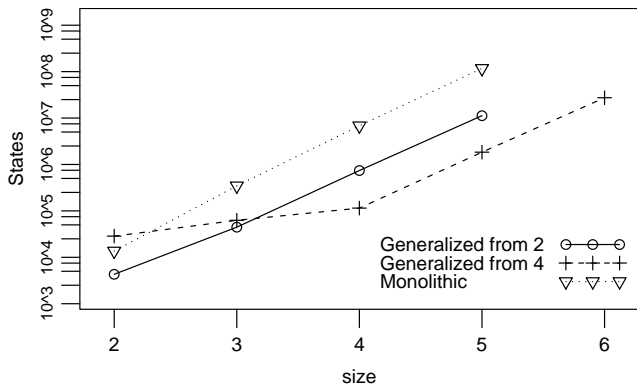


Figure 3: States Explored on the “Mutual Exclusion” Property of Gas Station with FLAVERS

investment of time.

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.

After encountering a number of two-way decompositions where it took more than a month to learn an assumption⁶, we imposed an upper bound on the amount of time we would spend examining a single two-way decomposition. This bound was set to be the maximum of 1 hour and 10 times the amount of time needed to verify that property monolithically⁷. 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 on 37.5% of the properties with FLAVERS and 36.7% 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 decom-

⁶The 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.

⁷On 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.

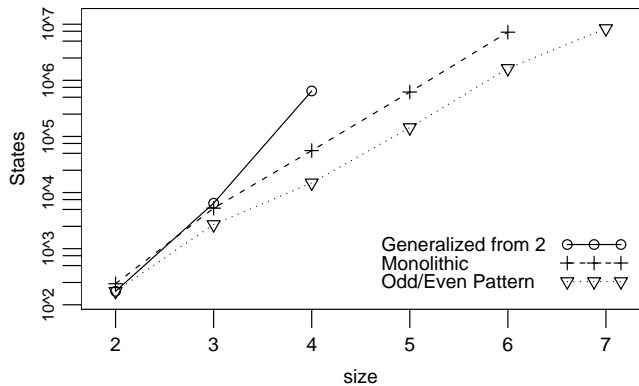


Figure 4: States Explored on the “Always 1 Between 0” Property of Relay with FLAVERS

position 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 was a pattern to the best decompositions. The pattern depended on whether or not the size of the system was 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 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 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 dur-

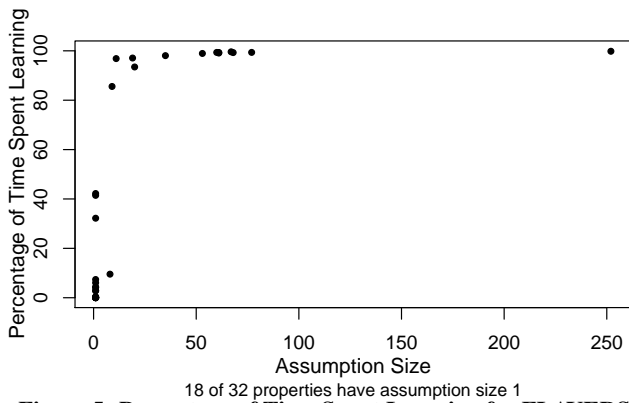


Figure 5: Percentage of Time Spent Learning for FLAVERS

ing 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. It is possible that compositional analysis without the L* algorithm could scale farther than compositional analysis with the L* algorithm. 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 decomposi-

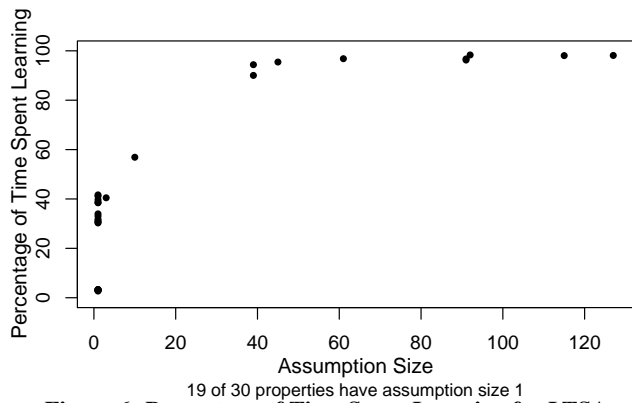


Figure 6: Percentage of Time Spent Learning for LTSA

tions 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, less 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 impossible 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 [16]. 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.

4. RELATED WORK

Our work uses the automated assume-guarantee approach of Cobleigh et al. [9], which uses the L^* algorithm to learn assumptions to complete assume-guarantee proofs. Several other assume-guarantee reasoning approaches based on the L^* algorithm have been proposed. The work of Barringer et al. [4] extends this approach to use symmetric assume-guarantee rules. Chaki et al. [5] developed an algorithm based on the L^* algorithm for learning tree automata for checking simulation conformance. The work of Giannakopoulou et al. [15] also computes assumptions, but requires exploring the entire state space of S_1 . 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. [18] have presented a framework for thread-modular abstraction refinement, in which assumptions and guarantees are both refined in an iterative fashion. 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 [12], but does not address notions of abstractions as is done in [18]. Jeffords and Heitmeyer use an invariant generation tool to generate invariants for components that can be used to complete an assume-guarantee proof [19]. 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., [13, 29]). 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., [7]) 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 [7], 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. Our experiments were limited in scope: we used one assume-guarantee reasoning technique, two finite-state verifiers, 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.

The results of our experiments are not very encouraging. The vast majority of decompositions explored more states than monolithic verification. If we restrict our attention to just the best decomposition at the smallest size for each example, then in only about half of the properties we examined did the assume-guarantee reasoning technique explore fewer states than monolithic verification. On the properties 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 better, 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 when assume-guarantee reasoning could save memory over monolithic verification, there were very few properties on which we could demonstrate that assume-guarantee reasoning could verify a larger system than monolithic verification. For the properties 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 properties 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 examining all two-way decompositions for larger systems sizes to find decompositions better than the generalized ones is too cost prohibitive to be useful in practice.

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. These negative results, although preliminary, raise doubts about the usefulness of assume-guarantee reasoning as an effective compositional analysis technique. Clearly additional experiments should be done using other automated learning techniques, other verification systems, and other decompositions approaches. Proposed advances in assume-guarantee reasoning, however, should now be accompanied by reasonable experimental evidence of effectiveness.

6. REFERENCES

- [1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.
- [2] 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.
- [3] 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.
- [4] H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proc. of the Second Work. on Spec. and Verification of Component-Based Systems*, pages 14–21, Sept. 2003.
- [5] S. Chaki, E. M. Clarke, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In K. Etessami and S. K. Rajamani, editors, *Proc. of the Seventeenth Int. Conf. on Computer-Aided Verification*, vol. 3576 of *LNCS*, pages 534–547, July 2005.
- [6] 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*, vol. 3083 of *LNCS*, pages 97–112, May 2004.
- [7] 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.
- [8] 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.
- [9] 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.*, vol. 2619 of *LNCS*, pages 331–346, Apr. 2003.
- [10] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd Int. Conf. on Soft. Eng.*, pages 439–448, June 2000.
- [11] 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.
- [12] C. Flanagan and S. Qadeer. Thread-modular model checking. In T. Ball and S. K. Rajamani, editors, *Proc. of the Tenth SPIN Work.*, vol. 2648 of *LNCS*, pages 213–224, May 2003.
- [13] 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.
- [14] D. Giannakopoulou and C. S. Păsăreanu. Learning-based assume-guarantee verification. In P. Godefroid, editor, *Proc. of the Twelfth SPIN Work.*, vol. 3639 of *LNCS*, pages 282–287, Aug. 2005.
- [15] 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.
- [16] 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.
- [17] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, Mar. 1985.
- [18] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. of the Fifteenth Int. Conf. on Computer-Aided Verification*, number 2725 in *LNCS*, pages 262–274. Springer-Verlag, July 2003.
- [19] R. D. Jeffords and C. L. Heitmeyer. A strategy for efficiently verifying requirements. In *Proc. of the Ninth European Soft. Eng. Conf./Eleventh ACM SIGSOFT Symp. on the Foundations of Soft. Eng.*, pages 28–37, Sept. 2003.
- [20] 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.
- [21] 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.
- [22] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
- [23] S. S. Patil. Limitations and capabilities of Dijkstra’s semaphore primitives for coordination among processes. Computational Structures Group Memo 57, Project MAC, Feb. 1971.
- [24] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.
- [25] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, vol. 13 of *NATO ASI*, pages 123–144. Springer-Verlag, Oct. 1984.
- [26] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, Apr. 1993.
- [27] 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.
- [28] 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.
- [29] 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.

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 .