Learning-based Symbolic Assume-guarantee Reasoning with Automatic Decomposition [⋆]

Wonhong Nam and Rajeev Alur

Dept. of Computer and Information Science
University of Pennsylvania
{wnam, alur}@cis.upenn.edu

Abstract. Compositional reasoning aims to improve scalability of verification tools by reducing the original verification task into subproblems. The simplification is typically based on the assume-guarantee reasoning principles, and requires decomposing the system into components as well as identifying adequate environment assumptions for components. One recent approach to automatic derivation of adequate assumptions is based on the L^* algorithm for active learning of regular languages. In this paper, we present a fully automatic approach to compositional reasoning by automating the decomposition step using an algorithm for hypergraph partitioning for balanced clustering of variables. We also propose heuristic improvements to the assumption identification phase. We report on an implementation based on NuSMV, and experiments that study the effectiveness of automatic decomposition and the overall savings in the computational requirements of symbolic model checking.

1 Introduction

To enhance the scalability of analysis tools, compositional verification suggests a "divide and conquer" strategy to reduce the verification task into simpler subtasks. The assume-guarantee based compositional reasoning to verify that a system S satisfies a requirement φ typically consists of the following three steps: (1) *System Decomposition*: decompose the system S into components $M_1, \cdots, M_n$, (2) *Assumption Discovery*: find an environment assumption A_i for each component M_i, and (3) *Assumption Checking*: verify that the assumptions A_i are adequate for proving or disproving the satisfaction of φ by S. The last step involves a number of verification subtasks, and while the exact nature of these subtasks depends on the specific compositional rule used, each subtask involves only one of the components M_i, and can be implemented using model checkers as it can be computationally less demanding than the original verification task.

The success of compositional reasoning depends on discovering appropriate assumptions for all the components so that the assumption checking phase will succeed, and one promising approach for automating this step is based on learning [9, 5, 3, 11]. If a component M_i communicates with its environment via a

[⋆] This research was partially supported by ARO grant DAAD19-01-1-0473, and NSF grants ITR/SY 0121431 and CCR0306382.

set X_i of boolean variables, then the assumption A_i can be viewed as a language over the alphabet 2^{X_i}, and the assumption checking constraints impose a lower and an upper bound on this language. The assumptions are constructed by adopting the L^* algorithm for learning a regular language using membership and equivalence queries [4, 17]. The membership query (whether a trace belongs to the desired assumption), and the equivalence query (whether the current assumption is adequate for the assumption checking phase) are implemented by invoking a model checker.

In this paper, we develop a fully automated framework for assume-guarantee based compositional reasoning by automating the decomposition phase also. While a modular description of a system can suggest a natural decomposition, an automated approach may be necessary for a variety of reasons: the description of a system, particularly when compiled from a high-level language to the input language of a model checker, is often monolithic; the decomposition suggested by the syntactic description need not be the one suitable for compositional reasoning, either in terms of the number of components or the partitioning of functionality among components. Our solution is based on an algorithm for partitioning of hypergraphs [14, 13]. Given a system S consisting of a set X of variables, and a desired number n of components, we partition the set X into n disjoint subsets $X_1, \cdots, X_n$ so that each set X_i contains approximately the same number of variables while keeping the number of communication variables (i.e. variables whose update depends on or affects a variable in another cluster) small. Each such variable partition X_i corresponds to a component M_i that controls these variables.

We describe an implementation of the automated compositional reasoning using parts of the state-of-the-art symbolic model checker NuSMV [7]. In our application, the alphabet size of the language being learnt itself grows exponentially with the number of communication variables. Consequently, in [3] we have developed a symbolic implementation of the L^* algorithm where the data structures for recording the membership information, and the assumption automaton, are maintained compactly using binary decision diagrams (BDDs) [6]. As described in Section 5, we have enhanced our implementation with several additional heuristics, in particular, one aimed at early falsification, and one aimed at deleting edges from the conjecture machine to force rapid convergence without violating the correctness of the learning algorithm.

In Section 6, we report on some examples where the original model contains around 100 variables, and the computational requirements of NuSMV are significant. The experiments are aimed at understanding the following tradeoffs: (1) how does our strategy for automatic decomposition compare with respect to the experiments we had performed earlier with manually chosen decomposition? (2) what is the impact of the number of components on the overall computational requirements? (3) how do the revised and more general assume-guarantee rule, and the new heuristics impact the performance? and (4) how does the integrated tool, *automatic symbolic compositional verifier* (ASCV), compare with NuSMV? It turns out the automatic decomposition strategy works pretty well, and manual (or structure-directed) decomposition seems unnecessary. No conclusions can be drawn regarding whether small or large number of components should be preferred in this approach. In terms of comparisons of the integrated tool with NuSMV, excellent gains are observed in some cases either reducing the required time or memory by two or three orders of magnitude, or converting infeasible problems into feasible ones. However, in some cases the number of states of the assumption is too large, and our learning-based strategy performs poorly.

Related Work While program slicing [20] is a technique to extract, from an original program, program statements relevant to a particular computation, compositional verification is to reduce a large verification problem into smaller subproblems. Compositional reasoning using assume-guarantee rules has a long history in formal verification literature (c.f. [19, 12, 1, 2, 16]). The use of learning algorithms for automatic discovery of assumptions was first reported in [9, 5], and has been further developed by many researchers: [18] considers the problem of substituting one component with another and how to reuse the conjecture machines computed in the original version while checking properties of the revised version; [8] reports several experiments to test whether assumptions with small DFA (deterministic finite automaton) representations exist. Our work is based on the symbolic implementation of learning-based compositional reasoning in [3]. The contributions of this paper include an automatic decomposition strategy, use of a more general assume-guarantee rule that is applicable to multiple components, heuristic improvements in computing the conjecture assumptions, and experiments to study several tradeoffs.

2 Preliminaries

We formalize the notions of a symbolic transition system and decomposition into its modules, and explain the assume-guarantee rule we use in this paper.

2.1 Symbolic transition systems

In the following, for any set of boolean variables X, we will denote the set of primed variables of X as $X' = \{x' \mid x \in X\}$. For a valuation q for X, q' denotes the valuation for X' such that $q'(x') = q(x)$ for every $x' \in X'$. A predicate $\varphi(X)$ is a boolean formula over X, and for a valuation q of variables in X, we write $\varphi(q)$ to mean that q satisfies the formula φ. We denote, given a predicate φ, a set of unprimed variables appearing in φ as $\mathit{Var}(\varphi)$.

A *symbolic transition system*, shortly a transition system, is a tuple $S(X, \mathit{Init}, T)$ with the following components:

- X is a finite set of boolean *variables*,
- $\mathit{Init}(X) = \bigwedge_{x \in X} \mathit{Init}_x(X)$ is an *initial predicate* over X, where $\mathit{Init}_x(X)$ is an initial predicate for the variable x,

- $T(X, X') = \bigwedge_{x \in X} T_x(X, X')$ is a *transition predicate* over $X \cup X'$ (X' represents a set of variables encoding the successor states), where $T_x(X, X')$ is a transition predicate for the variable x.

A *state* q of S is a valuation of the variables in X; i.e. $q : X \to \{\mathit{true}, \mathit{false}\}$. Let Q denote the set of all states q of S. For a state q over a set X of variables, let $q[Y]$, where $Y \subseteq X$ denote the valuation over Y obtained by restricting q to Y. The semantics of a transition system is defined in terms of the set of runs it exhibits. A *run* of $S(X, \mathit{Init}, T)$ is a sequence $q_0 q_1 \cdots$ where every $q_i \in Q$, such that $\mathit{Init}(q_0)$ holds, and for every $i \geq 0$, $T(q_i, q'_{i+1})$ holds. A *safety property* for a transition system $S(X, \mathit{Init}, T)$ is a predicate over X. For a transition system $S(X, \mathit{Init}, T)$ and a safety property $\varphi(X)$, we define $S \models \varphi$ if, for each run $q_0 q_1 \cdots$ of S, $\varphi(q_i)$ holds for each $i \geq 0$. Finally, given a transition system $S(X, \mathit{Init}, T)$ and a safety property $\varphi(X)$, an *invariant checking problem* is to check $S \models \varphi$.

2.2 Decomposition into modules

A *module* is a tuple $M(X_M, I_M, O_M, \mathit{Init}_M, T_M)$ with the following components:

- X_M is a finite set of boolean variables controlled by the module M,
- I_M is a finite set of boolean *input variables* that the module reads from its environment; I_M is disjoint from X_M,
- $O_M \subseteq X_M$ is a finite set of boolean *output variables* that are observable to the environment of M; let IO_M denote $I_M \cup O_M$,
- $\mathit{Init}_M(X_M, I_M)$ is an initial predicate over $X_M \cup I_M$,
- $T_M(X_M, I_M, X'_M)$ is a transition predicate over $X_M \cup I_M \cup X'_M$.

Given modules $M_1, \cdots, M_n$, where each $M_i = (X_{M_i}, I_{M_i}, O_{M_i}, \mathit{Init}_{M_i}, T_{M_i})$, we can compose them if for every i, X_{M_i} is disjoint from $X_{M_j} (j \neq i)$. We denote this composition as $M_1 \| \cdots \| M_n$, and a set of all input variables and output variables as IO (i.e. $\mathit{IO} = \bigcup_i \mathit{IO}_{M_i}$). As a symbolic transition system, the semantics for $M_1 \| \cdots \| M_n$ is defined in terms of the set of runs it exhibits. A *run* of $M_1 \| \cdots \| M_n$ is a sequence $q_0 q_1 \cdots$, where each q_i is a state over $X_{M_1} \cup \cdots \cup X_{M_n}$, such that for every $1 \leq j \leq n$, $\mathit{Init}_{M_j}(q_0[X_{M_j}], q_0[I_{M_j}])$ holds and for every $i \geq 0$ and $1 \leq j \leq n$, $T_{M_j}(q_i[X_{M_j}], q_i[I_{M_j}], q'_{i+1}[X'_{M_j}])$ holds. Again, given a composition of modules $M_1 \| \cdots \| M_n$ and a safety property φ over $X_{M_1} \cup \cdots \cup X_{M_n}$, we define $M_1 \| \cdots \| M_n \models \varphi$ if for every run $q_0 q_1 \cdots$ of $M_1 \| \cdots \| M_n$, $\varphi(q_i)$ holds for every $i \geq 0$.

Given a symbolic transition system $S(X, \mathit{Init}, T)$ and a set $Y \subseteq X$ of variables, we define a module $M[S, Y]$, shortly $M[Y]$, as a tuple $(X_{M[Y]}, I_{M[Y]}, O_{M[Y]}, \mathit{Init}_{M[Y]}, T_{M[Y]})$ as follows:

- $X_{M[Y]} = Y$,
- $\mathit{Init}_{M[Y]} = \bigwedge_{x \in Y} \mathit{Init}_x(X)$ where each $\mathit{Init}_x(X)$ is acquired from S,
- $T_{M[Y]} = \bigwedge_{x \in Y} T_x(X, X')$ where each $T_x(X, X')$ is also obtained from S,
- $I_{M[Y]} = \{x \in X \setminus Y \mid x \in \mathit{Var}(\mathit{Init}_{M[Y]}) \cup \mathit{Var}(T_{M[Y]})\}$,
- $O_{M[Y]} = \{x \in Y \mid \exists y \in X \setminus Y.\ x \in \mathit{Var}(\mathit{Init}_y) \cup \mathit{Var}(T_y)\}$.

Now, we can decompose a transition system $S(X, Init, T)$ into modules $M[X_1]$, $\cdots, M[X_n]$ by partitioning X into $X_1, \cdots, X_n$ where $X = \bigcup_i X_i$ and every X_i is disjoint from each other. In addition, we can denote this decomposition as $S \overset{dec}{=} M[X_1] \| \cdots \| M[X_n]$ using the composition operator $\|$, since every X_i is disjoint from each other. For the sake of simplicity, we will use $=$ instead of $\overset{dec}{=}$.

For a transition system $S(X, Init, T)$ decomposed into $M[X_1], \cdots, M[X_n]$ where each $M[X_i] = (X_{M[X_i]}, I_{M[X_i]}, O_{M[X_i]}, Init_{M[X_i]}, T_{M[X_i]})$, each run of S is obviously a run of $M[X_1] \| \cdots \| M[X_n]$ and each run of $M[X_1] \| \cdots \| M[X_n]$ is also a run of S, since $X = \bigcup_i X_i$, and $Init$ and T of S are equivalent to the conjunction of every $Init_{M[X_i]}$ and $T_{M[X_i]}$, respectively. Finally, given $S(X, Init, T)$ and a partition of X into disjoint subsets $X_1, \cdots, X_n$, $M[X_1] \| \cdots \| M[X_n] \models \varphi$ iff $S \models \varphi$.

2.3 Assume-guarantee rule

Given a module $M(X_M, I_M, O_M, Init_M, T_M)$, a *run* of M is, similarly with a run of a transition system, a sequence $q_0 q_1 \cdots$ where every q_i is a state over $X_M \cup I_M$ such that $Init(q_0[X_M], q_0[I_M])$ holds and for every $i \geq 0$, $T(q_i[X_M], q_i[I_M], q'_{i+1}[X'_M])$ holds. For a run $q_0 q_1 \cdots$ of M, the *trace* is a sequence $q_0[IO_M] q_1[IO_M] \cdots$. Let us denote the set of all the traces of M as $L(M)$, and the complement of the set as $L^C(M)$ (formally, $L^C(M) = Q_M^{IO^*} \setminus L(M)$ where Q_M^{IO} is a set of all the states over IO_M). For a trace set L over a variable set IO_M and a safety property φ over IO_M, we can extend the notion of $\models$ to trace sets as following: $L \models \varphi$ if, for every trace $q_0 q_1 \cdots \in L$, $\varphi(q_i)$ holds for every $i \geq 0$. In addition, the composition operator $\|$ can be extend to trace sets which have the same alphabet (i.e. the same set of input/output variables) as following: for L_1 and L_2 with the same I/O variable set, $L_1 \| L_2 = L_1 \cap L_2$.

Now, we use the following assume-guarantee rule to prove that a composition of modules, $M_1 \| \cdots \| M_n$ satisfies a safety property φ over IO where for every module A_i, IO_{A_i} equals to IO of $M_1 \| \cdots \| M_n$ ($IO = \bigcup_i IO_{M_i}$).

$$\frac{\begin{array}{rl} M_1 \| A_1 \models \varphi, \cdots, M_n \| A_n \models \varphi & \quad (\mathbf{Pr1}) \\ L^C(A_1) \| \cdots \| L^C(A_n) \models \varphi & \quad (\mathbf{Pr2}) \end{array}}{M_1 \| \cdots \| M_n \models \varphi}$$

The rule above says that if there exist assumption modules $A_1, \cdots, A_n$ such that for each i, the composition of M_i and A_i is safe (i.e. satisfies the property φ) and the composition of the complements of every A_i satisfies φ, then $M_1 \| \cdots \| M_n$ satisfies φ. Intuitively, the first premise **Pr1** makes every assumption strong enough to keep each M_i safe, and the second premise **Pr2** makes the assumptions weak enough to cover all the traces which can violate φ (i.e., for every trace violating φ, **Pr2** requires at least one assumption to contain it). This rule is sound and complete [5]. Our aim is to construct such assumptions $A_1, \cdots, A_n$ to show that $M_1 \| \cdots \| M_n$ satisfies φ, and the smaller assumptions can save the more in terms of searching state space.

Given a symbolic transition system $S(X, \mathit{Init}, T)$, an integer $n \geq 2$ and a safety property φ, the *model-checking problem* we consider in this paper is, instead of checking $S \models \varphi$, to partition X into disjoint subsets $X_1, \cdots, X_n$, and to check $M[X_1] \| \cdots \| M[X_n] \models \varphi$ using the above assume-guarantee rule. Note that we are assuming that the safety property φ is a predicate over IO, but this is not a restriction: to check a property that refers to private variables of a module, we can simply declare them as output variables. Finally, the challenges of this paper are (1) how to find a variable partition and (2) how to find assumptions satisfying both of the above premises.

3 Automatic partitioning

Automatic partitioning is, given a transition system $S(X, \mathit{Init}, T)$ and an integer $n \geq 2$, to decompose X into disjoint subsets $X_1, \cdots, X_n$, and there exist about $n^{|X|}$ possible partitions. Among them, we want a partition to minimize memory usage for assumption construction and commitment in our assume-guarantee reasoning. The memory usage, however, cannot be formulated. Therefore, we roughly fix our goal to find a partition that has small number of variables required in each step of the assume-guarantee reasoning because a state space for each step is exponential in the number of variables. More precisely, the alternative goal is to find a partition that minimizes $max_i(|X_i \cup IO_{M_i}|)$ where IO_{M_i} is the set of I/O variables of module $M[X_i]$. This partitioning problem is NP-complete.

We reduce our problem into a well-known partitioning problem called the *hypergraph partitioning problem* which can be used for directed-graph partitioning. For the reduction, we relax our goal as following; given a transition system $S(X, \mathit{Init}, T)$, and an integer $n \geq 2$, our automatic partitioning is to find a partition decomposing X into n disjoint subsets such that (1) the number of variables in each module is in some bound (near even distribution) and (2) modules corresponding to each variable subset have as few input/output variables as possible.

A *hypergraph* $G(V, E)$ is defined as a set of vertices V and a set of *hyperedges* E where each hyperedge is a set of arbitrary number of vertices in V. Thus, an ordinary graph is a special case of hypergraphs such that every edge is a pair of two vertices. Given a hypergraph $G(V, E)$ and an overall load imbalance tolerance $c \geq 1.0$, the *k-way hypergraph partitioning problem* is to partition the set V into k disjoint subsets, $V_1, \cdots, V_k$ such that the number of vertices in each set V_i is bounded by $|V|/(c \cdot k) \leq |V_i| \leq |V|(c/k)$, and the size of *hyperedge-cut* of the partition is minimized where the hyperedge-cut is a set of hyperedges e such that there exist v_1 and v_2 in e which belong to different partitions.

Now, our partitioning problem can be reduced to the k-way hypergraph partitioning problem. Given a transition system $S(X, \mathit{Init}, T)$, we construct a hypergraph $G(V, E)$ as follows. $V = \{v_x \mid x \in X\}$. For each $x \in X$, we have a hyperedge e_x that immediately contains the corresponding vertex v_x and also vertices v_y such that $x \in \mathit{Var}(\mathit{Init}_y) \cup \mathit{Var}(T_y)$. Intuitively, e_x represents the corresponding variable x and all the variables to read x. Finally, E is the set of all

e_x. Then, after hypergraph partitioning, $V_1, \cdots, V_k$ correspond with $X_1, \cdots, X_n$ in our problem. If we have a hyperedge e_x in the hyperedge-cut (let us assume that the corresponding vertex v_x belongs to V_i), then there exist some vertex $v_y \in e_x$ which belongs to $V_j (i \neq j)$. Since y is dependent on x but they are in different partitions, x should be an input variable of $M[X_j]$ and also an output variable of $M[X_i]$. For the overall load imbalance tolerance c, a large value for c can reduce the number of I/O variables but it causes larger imbalance among each module. On the other hand, a small value for c increases I/O variables. Therefore, we perform partitioning with six different values (i.e. 1.0, 1.2, $\cdots$, 2.0) and pick the partition that has the minimum value as $max_i(|X_i \cup IO_{M_i}|)$.

Many researchers have studied this problem and developed tools, and among them we use hMETIS [14]. hMETIS is one of the state-of-the-art hypergraph partitioning tools which uses a multilevel k-way partitioning algorithm. The multilevel partitioning algorithm has three phases; (1) it first reduces the size of a given hypergraph by collapsing vertices and edges until the hypergraph is small enough (*coarsening phase*), (2) the algorithm partitions it into k sub-hypergraphs (*initial partitioning phase*), and (3) the algorithm uncoarsens them to construct a partition for the original hypergraph (*uncoarsening and refinement phase*). Experiments on a large number of hypergraphs arising in various domains including VLSI, databases and data mining show that hMETIS produces partitions that are consistently better than those produced by other widely used algorithms, such as KL [15] and FM [10]. In addition, it is so fast as to produce high quality bisections of hypergraphs with 100,000 vertices in 3 minutes [13].

4 Learning assumptions

In this section, we define the *weakest safe assumption tuple* which is a witness for the truth of a given invariant, and briefly explain an algorithm for learning regular languages, called *L^* algorithm.* We then establish that our verification algorithm based on the L^* algorithm converges to the weakest safe assumption tuple or, before that, concludes with a witness for the invariant.

4.1 Weakest safe assumptions

After partitioning, our aim is, given a set of modules $M[X_1], \cdots, M[X_n]$ (obtained from automatic partitioning) and a safety property $\varphi(IO)$, to verify that $M[X_1] \| \cdots \| M[X_n] \models \varphi$ by finding assumption modules $A_1, \cdots, A_n$ that satisfy both premises of our assume-guarantee rule. A tuple $(A_1, \cdots, A_n)$ of assumptions is called a *safe assumption tuple* (ST) if the assumptions $A_1, \cdots, A_n$ satisfy **Pr1**, and a tuple $(A_1, \cdots, A_n)$ of assumptions is called an *appropriate assumption tuple* (AT) if the assumptions $A_1, \cdots, A_n$ satisfy both of **Pr1** and **Pr2**. For every $M[X_i]$, the *weakest safe assumption* W_i is a module such that $M[X_i] \| W_i \models \varphi$ and $L(W_i) \supseteq L(A_i)$ for every A_i such that $M[X_i] \| A_i \models \varphi$. We denote such a tuple $(W_1, \cdots, W_n)$ as the *weakest safe assumption tuple* (WT). Now, we show that the WT is a witness for the truth of $M[X_1] \| \cdots \| M[X_n] \models \varphi$.

Lemma 1. *If* $M[X_1]\|\cdots\|M[X_n] \models \varphi$*, the WT* $(W_1,\cdots,W_n)$ *is a witness of* $M[X_1]\|\cdots\|M[X_n] \models \varphi$*.*

Proof. If $M[X_1]\|\cdots\|M[X_n]$ does indeed satisfy φ, then there exists an AT $(A_1,\cdots,A_n)$ since the composition rule is complete. By definition, $(W_1,\cdots,W_n)$ satisfies **Pr1**. For the above AT $(A_1,\cdots,A_n)$, since for every i, $L^C(W_i) \subseteq L^C(A_i)$ and $L^C(A_1)\|\cdots\|L^C(A_n) \models \varphi$, $L^C(W_1)\|\cdots\|L^C(W_n) \models \varphi$ (**Pr2**). Finally, the WT $(W_1,\cdots,W_n)$ is one of ATs and a witness of $M[X_1]\|\cdots\|M[X_n] \models \varphi$.

Lemma 2. *If* $M[X_1]\|\cdots\|M[X_n] \not\models \varphi$*, the WT* $(W_1,\cdots,W_n)$ *is a witness of* $M[X_1]\|\cdots\|M[X_n] \not\models \varphi$*.*

Proof. If $M[X_1]\|\cdots\|M[X_n]$ does not satisfy φ, then there is no AT; i.e., if an assumption tuple $(A_1,\cdots,A_n)$ satisfies **Pr1**, there exists a trace $\tau \in L^C(A_1)\|\cdots\| L^C(A_n)$ violating φ. Again, since $(W_1,\cdots,W_n)$ satisfies **Pr1** by definition, there exists $\tau \in L^C(W_1)\|\cdots\|L^C(W_n)$ violating φ. For every $(A_1,\cdots,A_n)$ that satisfies **Pr1**, since for every i, $L^C(W_i) \subseteq L^C(A_i)$ and $L^C(W_1)\|\cdots\|L^C(W_n) \subseteq L^C(A_1)\|\cdots\|L^C(A_n)$, the above trace τ violating φ also belongs to $L^C(A_1)\|\cdots\| L^C(A_n)$. Thus, the WT $(W_1,\cdots,W_n)$ is a witness of $M[X_1]\|\cdots\|M[X_n] \not\models \varphi$.

The WT $(W_1,\cdots,W_n)$ can be represented by a tuple of DFAs with the alphabet Q^{IO} (where Q^{IO} is a set of all states over IO) as each $M[X_i]$ is finite. Therefore, we can learn the WT which a witness for truth of $M[X_1]\|\cdots\|M[X_n] \models \varphi$, using the L^* algorithm for learning regular languages.

4.2 L^* algorithm

The L^* *algorithm* learns an unknown regular language U (let Σ be its alphabet) and generates a minimal DFA that accepts the regular language. This algorithm was introduced by Angluin [4], but we use an improved version by Rivest and Schapire [17]. The algorithm infers the structure of the DFA by asking a teacher, who knows the unknown language, membership and equivalence queries. Membership queries ask whether a given string $\sigma \in \Sigma^*$ is in the language U, and the answer for the queries is yes or no. Equivalence queries ask whether a given conjecture DFA C represents the language U, and the answer is yes or no with a counter-example that is a symmetric difference between $L(C)$ and U.

At any given time, the L^* algorithm has, in order to construct a conjecture machine, information about a finite collection of strings over Σ, classified either as members or non-members of U based on membership queries. This information is maintained in an *observation table* (Rs, Es, Mp) which represents the conjecture DFA; Rs is a set of representative strings for states in the DFA such that each representative string $r_q \in Rs$ for a state q leads from the initial state (uniquely) to the state q, and Es is a set of experiment suffix strings that are used to distinguish states. Mp maps strings σ in $(Rs \cup Rs\cdot\Sigma)\cdot Es$ to 1 if σ is in U, and to 0 otherwise. Once a conjecture machine C is built, the algorithm asks an equivalence query. Finally, if the answer is 'yes', it returns the current conjecture DFA C; otherwise, a counter-example $cex \in ((L(C) \setminus U) \cup (U \setminus L(C))$

is provided by the teacher. In the latter case, the algorithm updates the current conjecture using the counter-example *cex*.

If a teacher for two kinds of queries is provided, the L^* algorithm is guaranteed to construct a minimal DFA for the unknown regular language using only $O(|\Sigma|n^2 + n \log m)$ membership queries and at most $n-1$ equivalence queries, where n is the number of states in the final DFA and m is the length of the longest counter-example provided by the teacher for equivalence queries.

4.3 Automatic symbolic compositional verification

Now, we present our verification algorithm. Given a transition system $S(X, \mathit{Init}, T)$, an invariant property φ, and an integer $n \geq 2$, our *automatic symbolic compositional verification* (ASCV) algorithm decomposes X into n disjoint subsets $X_1, \cdots, X_n$ and then checks $M[X_1] \| \cdots \| M[X_n] \models \varphi$ by learning the WT (weakest safe assumption tuple), which is a witness for the truth of the invariant. For learning the WT, the ASCV algorithm provides teachers who answer membership and equivalence queries, which correspond with the WT $(W_1, \cdots, W_n)$.

Given a string $\tau \in Q^{IO^*}$ and a module $M[X_i]$, a teacher for membership queries answers whether there is an execution of $M[X_i]$ consistent with τ, which violates φ; that is, whether $\tau \in L(W_i)$. The ASCV algorithm constructs a conjecture assumption A_i for each module $M[X_i]$, based on the results of membership queries, and after this phase, it asks an equivalence query. The equivalence query consists of two sub-queries: checking **Pr1** and **Pr2** of the assume-guarantee rule. If a given assumption tuple satisfies both premises, we conclude $S = M[X_1] \| \cdots \| M[X_n] \models \varphi$; otherwise, the teacher produces a counter-example. More precisely, the teacher checking **Pr1** answers, given an assumption A_i for a module $M[X_i]$, whether $M[X_i] \| A_i \models \varphi$; if not, it returns $\tau \in L(A_i)$ violating φ (i.e. $\tau \in L(A_i) \setminus L(W_i)$). The teacher for **Pr2** checks, given $A_1, \cdots, A_n$, whether $L^C(A_1) \| \cdots \| L^C(A_n) \models \varphi$; if not, it returns $\tau \in L^C(A_i)$ for every i which violates φ. For **Pr1** queries, τ is immediately used to update A_i, but for **Pr2** queries, we need an additional analysis. That is, when we execute every $M[X_i]$ corresponding to τ, if every $M[X_i]$ reaches a state violating φ, then τ is a counter-example of the original problem, $S = M[X_1] \| \cdots \| M[X_n] \models \varphi$; otherwise, τ is used to update A_i such that $M[X_i]$ correspondent with A_i does not violate the invariant φ (i.e. $\tau \in L(W_i) \setminus L(A_i)$).

In the ASCV algorithm, since all answers from teachers are always consistent with the WT (for equivalence queries, counter-examples are checked with each W_i), our ASCV algorithm will converge to the WT which a witness for the truth of $S \models \varphi$, in the polynomial number of queries by the property of the L^* algorithm. However, there can be early termination with a counter-example or an AT satisfying both premises. In addition, the algorithm will not generate any assumption A_i with more states than W_i.

Figure 1 illustrates our ASCV algorithm. Given a transition system S, a safety property φ, and an integer n, the ASCV algorithm first decomposes S into n modules and assigns them to an array $M[\,]$ (line 1), and it constructs the initial conjecture machines according to the rule of the L^* algorithm (line

```
    Boolean ASCV($S, \varphi, n$)
1:  $M[\,]$ := AutomaticPartitioning($S, \varphi, n$);
2:  $A[\,]$ := InitializeAssumptions($M[1], \cdots, M[n], \varphi$);
3:  repeat:
4:      foreach($1 \le i \le n$){
5:          while(($cex$ := SafeAssumption($M[i], A[i], \varphi$)) $\neq$ $null$){
6:              UpdateAssumption($M[i], A[i], cex$);
7:      } }
8:      if(($cex$ := DischargeAssumptions($A[1], \cdots, A[n], \varphi$)) = $null$){
9:          return $true$;
10:     } else {
11:         $IsRealCex$ := $true$;
12:         foreach($1 \le i \le n$) {
13:             if(SafeTrace($M[i], cex$)) {
14:                 UpdateAssumption($M[i], A[i], cex$);
15:                 $IsRealCex$ := $false$;
16:         } }
17:         if($IsRealCex$) return $false$;
18:     }
```

Fig. 1. Automatic symbolic compositional verification algorithm

2). Then, we repeat asking two sub-queries for equivalence and updating the current conjecture machines; if either of them returns a counter-example *cex*, the algorithm updates the conjecture machines using *cex* (lines 4–18). In more detail, we check that for every i, the current $A[i]$ is a safe assumption such that $M[i] \| A[i] \models \varphi$ by a function `SafeAssumption()`. If so, we have $A[1], \cdots, A[n]$ satisfying **Pr1**; otherwise (i.e., for some i, we have a counter-example *cex*), we update $A[i]$ with respect to *cex* (line 6). Once we have $A[1], \cdots, A[n]$ satisfying **Pr1**, the algorithm checks **Pr2** by a function `DischargeAssumptions()`. If the function returns *null*, then we conclude $S \models \varphi$ since $A[1], \cdots, A[n]$ satisfy both premises; otherwise, we are provided a counter-example *cex*. Lines 11–17 analyze whether *cex* is a real counter-example for the invariant; if *cex* indeed violates φ for every $M[i]$, then we conclude $S \not\models \varphi$. Otherwise, it is a spurious counter-example and we update $A[i]$ that is the conjecture for $M[i]$ not violating φ.

5 Symbolic implementation

The ASCV algorithm can be implemented explicitly as well as implicitly. However, as input/output variables increase, the number of the alphabet symbols of the languages we want to learn also increases exponentially. In explicit implementations [9, 11], the large alphabet size poses crucial problems: (1) the constructed assumption DFAs have too many edges when represented explicitly, (2) the size of the observation tables for each assumption gets very large, and (3) the number of membership queries needed to fill each entry in the observation tables also increases. In [3] we introduced a symbolic implementation for learning-based compositional verification and we, in this paper, extend the technique.

5.1 Data structures and functions

For symbolic implementation, we already defined a symbolic transition system and decomposition to modules implicitly in Section 2. Here, we present the rest of important symbolic data structures used in the ASCV algorithm.

- Each conjecture assumption A_i is also a module $A_i(X_{A_i}, I_{A_i}, O_{A_i}, \mathit{Init}_{A_i}, T_{A_i})$ that can be constructed using BDDs. Each A_i represents a conjecture DFA in the L^* algorithm: X_{A_i} encodes a set of states, IO_{A_i} represents its alphabet, and Init_{A_i} and T_{A_i} encode an initial state and a transition function, respectively.
- Observation table $(\mathit{Rs}, \mathit{Es}, \mathit{Mp})$ for each conjecture assumption A_i is maintained using BDDs. Each representative string $r \in \mathit{Rs}$ is encoded by a BDD representing a set of states of $M[X_i]$ reachable by r (i.e. $\mathit{PostImage}(\mathit{Init}_{M[X_i]}, r)$). Every experiment string $e \in \mathit{Es}$ is also represented by a BDD encoding a set of states of $M[X_i]$ from which some state violating φ is reachable by e (i.e. $\mathit{PreImage}(\neg\varphi, e)$). Mp is maintained by a set of boolean arrays.
- A counter-example *cex* is a finite sequence of states over IO, and it is represented by a list of BDDs.

All functions in the ASCV algorithm are implemented using symbolic computation as following (where all the parameters are already represented by BDDs).

- `SafeAssumption`$(M[X_i], A_i, \varphi)$ checks $M[X_i] \| A_i \models \varphi$. It can be achieved by an ordinary symbolic reachability test.
- `DischargeAssumptions`$(A_1, \cdots, A_n, \varphi)$ checks $L^C(A_1) \| \cdots \| L^C(A_n) \models \varphi$. For every A_i, we first construct a module encoding a complement DFA of A_i. This complementing can be easily performed even in our symbolic implementation. We then check that the composition of the complement DFAs satisfies φ, which is also handled by the symbolic reachability test.
- `UpdateAssumption`$(M[X_i], A_i, \mathit{cex})$ reconstructs the conjecture assumption A_i for the module $M[X_i]$ to be used in the next iteration. It first finds a new experiment string that is the longest suffix of *cex* which can demonstrate the difference between the current conjecture and the goal language. We then update the observation table for A_i by adding the new experiment string. This addition introduces new states and edges. We identify a set of edges between states by BDD computation.
- `SafeTrace`$(M[X_i], \mathit{cex})$ checks, by the reachability test, that there exists any trace of $M[X_i]$ corresponding with *cex*, which violates φ.

5.2 Early falsification

In the previous implementations of learning-based compositional verification [9, 11] including ours [3], we have found a possible optimization that allows us to conclude earlier $S \not\models \varphi$ with a counter-example. In Figure 1, if *cex* acquired from `DischargeAssumptions()` reaches some state violating φ for every $M[X_i]$, then we conclude that the invariant is false (line 17). That is, in the case that the invariant is indeed false, the algorithm cannot terminate until encountering safe assumptions for each module and checking `DischargeAssumptions()`. On the other hand, *cex* provided from `SafeAssumption()` is immediately used for updating the current conjecture (line 6) even though it is a candidate of evidence for $S \not\models \varphi$. In our new implementation, if *cex* obtained from `SafeAssumption()` is a feasible trace for every other module $M[X_j](j \neq i)$, then we declare *cex* as a counter-example for $S \models \varphi$. Otherwise (*cex* violates φ in $M[X_i]$, but it is infeasible for some other module), we update the current assumption for $M[X_i]$ to rule out *cex* as the original algorithm. We believe that the additional feasibility checking adds a little effort in terms of time and memory, but sometimes this function can falsify the invariant earlier. We will present examples where we can conclude much earlier than experiments without *early falsification* in Section 6. The function `EarlyFalsify()` is implemented as below:

```
EarlyFalsify(Trace $\tau$, int $MNum$){
    foreach $(j \neq MNum)$
        if $(\neg$ FeasibleTrace$(M[j], \tau))$ return $false$;
    return $true$;
}
```

Finally, we add the function `EarlyFalsify()` between line 5 and 6 in the ASCV algorithm (see Figure 1).

```
5:     while(($cex :=$ SafeAssumption$(M[i], A[i], \varphi)) \neq null$){
5′:        if(EarlyFalsify$(cex, i)$) return $false$;
6:         UpdateAssumption$(M[i], A[i], cex)$;
```

5.3 Edge deletion for safe assumptions

The ultimate goal of our model-checking problem is to quickly discover a small AT (appropriate assumption tuple) or a counter-example for $S \models \varphi$. The ASCV algorithm, however, only guarantees that we can eventually learn the WT (weakest safe assumption tuple) whose size is, in theory, exponential in the size of each module in the worst case. That is, the ASCV algorithm based on the L^* algorithm may keep introducing new states for conjecture machines until converging on a very large WT, even though there may exist smaller ATs than the WT. We have experienced many cases where our algorithm needs many iterations to converge on the WT (lines 5–6). The optimal solution for this problem is to learn the smallest AT in terms of the number of states rather than the WT, but this is a computationally hard problem.

Instead, we propose a simple heuristic called *edge deletion* for this problem where we retry, without introducing new states, to check **Pr1** and **Pr2** after eliminating some edges from the current assumption. More precisely, when we are given a counter-example *cex* from `SafeAssumption`$(M[X_i], A_i, \varphi)$, *cex* is a list of BDDs encoding a set of counter-examples to reach some state violating φ. Each counter-example is a sequence of states of $M[X_i] \| A_i$, and we can extract the edge of A_i from the last transition of the sequence which immediately leads to the state violating φ. By disallowing the edges from A_i, we can rule out *cex* from the current conjecture machine A_i. Then, we check `SafeAssumption()` again; if we get a safe assumption by the retrial, we proceed to the next step. If we cannot conclude using this stronger assumption, then we replace it with the original assumption and update the original one for the next iteration. This replacement ensures the convergence to the WT. Intuitively, our heuristic *edge deletion* searches, with the same number of states, more broadly in solution candidate space, while the original L^* algorithm keeps searching deeply by introducing new states. We believe that sometimes this heuristic also can encounter a smaller AT than the original algorithm. Section 6 shows evidence of this benefit.

6 Experiments

We have implemented our automatic symbolic compositional verification algorithm with the BDD package in a symbolic model checker NuSMV. For experiment, we have six sets of examples where five sets are collected from the NuSMV package and one is artificial. For the artificial examples, we know that small assumptions exist, and for examples from NuSMV package, we added some variables or scaled them up as tools finished fast with the original models. All experiments have been performed on a Sun-Blade-1000 workstation using a 750MHz UltraSPARC III processor, 1GB memory and SunOS 5.10. First, we compare our *automatic symbolic compositional verifier* (ASCV) with our previous implementation in [3]. We then present effects of the number of partitions and new features (early falsification and edge deletion in Section 5). Finally, we compare our ASCV with the invariant checking (with early termination) of NuSMV 2.3.0. Each result table has the number of variables in total (tv), I/O variables, $max_i(|X_i \cup IO_{M_i}|)$ (mx), execution time in seconds, the peak BDD size and the number of states in the assumptions we learn (asm). The running time includes time to perform partitioning as well. Entries denoted by '–' mean that a tool did not finish within 2 hours. In addition, columns denoted by 'F/D' mean that early falsification or edge deletion contributes to concluding earlier, and 'np' means the number of partitions.

ASCV vs. SCV Compared with the previous implementation SCV in [3], ASCV has the following new features: automatic partitioning, a symmetric compositional rule, early falsification and edge deletion. Table 1 presents that ASCV shows better performance in 10 over 14 examples. However, since the examples in Table 1 were selected in [3] so as to explain that SCV worked well, they may be favorable to SCV. Also, in all the examples, automatic partitioning is as good as manual partitioning in terms of $max_i(|X_i \cup IO_{M_i}|)$, and in 5 cases it reduces the numbers by 20–40%.

The number of partitions Table 2 shows how the number of partitions affects the performance. In two cases, increasing the number of partitions saves significantly in terms of time and BDD usage by keeping generating small assumptions. However, other two cases need more time and BDDs due to large assumptions. Therefore, one has to experiment with the different number of partitions for better results.

example name	spec	tot var	SCV				ASCV					
			mx	IO	time	peak BDD	np	mx	IO	time	peak BDD	F/D
simple1	true	69	37	4	19.2	607,068	2	36	4	4.9	605,024	D
simple2		78	42	5	106	828,842	2	41	5	31.3	620,354	D
simple3		86	46	5	754	3,668,980	2	46	5	223	2,218,762	D
simple4		94	50	5	4601	12,450,004	2	50	5	1527	9,747,836	D
guidance1	false	135	118	23	124	686,784	2	89	18	–	–	–
guidance2	true	122	105	22	196	1,052,660	4	59	18	6.6	359,744	D
guidance3	true	122	93	46	357	619,332	2	76	15	–	–	–
barrel1	false	60	35	10	20.3	345,436	2	35	10	–	–	–
barrel2	true	60	35	10	23.4	472,164	2	35	10	–	–	–
msi1	true	45	37	25	2.1	289,226	2	37	19	0.3	50,078	D
msi2		57	49	25	37.0	619,332	2	49	22	1.8	524,286	D
msi3		70	62	26	1183	6,991,502	2	60	25	31.9	2,179,926	D
robot1	false	92	89	12	1271	4,169,760	2	52	5	283	1,905,008	F
robot2	true	92	75	12	1604	2,804,368	2	50	7	9.5	427,196	D

Table 1. Comparison between SCV and ASCV

np	simple4							guidance2						
	spec	tv	mx	IO	time	peak BDD	asm	spec	tv	mx	IO	time	peak BDD	asm
2	true	94	49	5	1526	9,747,836	2,2	true	122	82	18	1680	612,178	2,2
3			61	37	1.8	497,714	2,2,2			61	23	34	614,222	2,2,2
4			53	37	0.7	217,686	2,2,2,2			59	33	6.6	359,744	2,2,2,2

np	robot1							syncarb4						
	spec	tv	mx	IO	time	peak BDD	asm	spec	tv	mx	IO	time	peak BDD	asm
2	false	92	52	5	283	1,905,008	3,2	true	21	21	21	332	7,700,770	131,131
3			62	30	–	–	too many			21	21	643	14,870,100	35,19,35
4			64	46	–	–	too many			21	21	4520	31,234,364	11,11,19,19

Table 2. Effect of the number of partitions

example name	spec	tot var	np	mx	IO var	Without F/D			With F/D			
						time	peak BDD	asm	time	peak BDD	asm	F/D
simple1	true	69	2	36	4	10.3	605,024	2,3	4.9	605,024	2,2	D
simple2		78	2	41	5	58.3	624,442	2,3	31.3	620,354	2,2	D
simple3		86	2	45	5	441	2,997,526	3,2	223	1,849,462	2,2	D
simple4		94	2	49	5	3044	9,747,836	2,3	1526	9,747,836	2,2	D
guidance2	true	122	2	105	18	1634	1,066,968	2,37	1603	612,178	2,2	D
msi1	true	45	2	37	19	–	–	too many	0.3	49,056	2,2	D
msi2		57	2	49	22	–	–	too many	1.8	524,286	2,2	D
msi3		70	2	60	25	–	–	too many	31.9	2,179,926	2,2	D
robot1	false	92	2	52	5	529	2,275,994	3,58	283	1,905,008	3,2	F
robot2	true	92	2	50	7	10.4	529,396	2,3	9.5	427,196	2,2	D
syncarb1	false	18	2	18	18	28.2	1,384,810	67,67	125	1,536,066	67,67	–
syncarb2	true	18	2	18	18	30.4	1,274,434	67,67	86.6	1,280,566	67,67	–

Table 3. With/without Early falsification and edge deletion

Early falsification & edge deletion In Table 3, we present how our new features help to conclude earlier. In case of that given invariants are true, our edge deletion heuristic saves the number of states of assumptions in many examples. In case of false, early falsification helps to save. In two examples (`syncarb1` and `syncarb2`), however, these features affect performance adversarially.

ASCV vs. NuSMV Finally, Table 4 presents the comparison between ASCV (with the heuristics) and NuSMV. In 10 examples, ASCV is significantly better than NuSMV where we have found small assumptions. In `syncarb3` and `syncarb4`, however, the assumptions we have learnt are relatively large (with more than 100 states for each) and we believe that the large size of assumptions

example	spec	tot	ASCV					NuSMV	
name		var	np	mx	IO	time	peak BDD	time	peak BDD
simple1	true	69	2	36	4	4.9	605,024	269	3,993,976
simple2		78	2	41	5	31.3	620,354	4032	32,934,972
simple3		86	4	50	37	1.0	330,106	–	–
simple4		94	4	53	37	0.7	217,686	–	–
guidance2	true	122	4	59	18	6.6	359,744	–	–
msi1	true	45	2	37	19	0.3	50,078	157	1,554,462
msi2		57	2	49	22	1.8	524,286	3324	16,183,370
msi3		70	2	60	25	31.9	2,179,926	–	–
robot1	false	92	2	52	5	283	1,905,008	654	2,729,762
robot2	true	92	2	50	7	9.5	427,196	1039	1,117,046
syncarb3	false	21	2	21	21	351	9,948,148	0.1	5,110
syncarb4	true	21	2	21	21	332	7,700,770	0.1	3,066
barrel1	false	60	–	–	–	–	–	1201	28,118,286
barrel2	true	60	–	–	–	–	–	4886	36,521,170

Table 4. Comparison between ASCV and NuSMV

is a main reason of negative results in these examples. Also, it can explain why ASCV cannot complete in the timeout in `barrel1` and `barrel2`. More details about the examples are available at `http://www.cis.upenn.edu/∼wnam/ASCV/`.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM TOPLAS*, 17:507–534, 1995.
2. R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. A preliminary version appears in *Proc. 11th LICS, 1996.*
3. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proc. CAV 2005*, pages 548–562, 2005.
4. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.
5. H. Barringer, C.S. Pasareanu, and D. Giannakopolou. Proof rules for automated compositional verification through learning. In *Proc. 2nd SVCBS*, 2003.
6. R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
7. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. CAV 2002*, LNCS 2404, pages 359–364, 2002.
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. *Technical Report UM-CS-2004-023*, 2005.
9. J.M. Cobleigh, D. Giannakopoulou, and C.S. Pasareanu. Learning assumptions for compositional verification. In *Proc. 9th TACAS*, LNCS 2619, pages 331–346, 2003.
10. C.M. Fiduccia, R.M. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. of 19th DAC*, pages 175–181, 1982.
11. D. Giannakopoulou, C.S. Pasareanu. Learning-based assume-guarantee verification. In *Proc. of SPIN 2005*, pages 282–287, 2005.
12. O. Grümberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
13. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Trans. VLSI Systems*, 7(1):69–79, 1999.
14. G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *In Proc. of 36th Design Automation Conference*, pages 343–348, 1999.
15. B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.
16. K.L. McMillan. A compositional rule for hardware design refinement. In *CAV 97: Computer-Aided Verification*, LNCS 1254, pages 24–35, 1997.
17. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
18. N. Sharygina, S. Chaki, E.M. Clarke, and N. Sinha. Dynamic component substitutability analysis. In *Proc. of FM 2005*, LNCS 3582, pages 512–528, 2005.
19. E.W. Stark. A proof technique for rely-guarantee properties. In *FST & TCS 85*, LNCS 206, pages 369–391, 1985.
20. M. Weiser. Program slicing. IEEE Trans. on Software Engineering, 10:352–357, 1984.