Form Methods Syst Des (2008) 32: 207–234
DOI 10.1007/s10703-008-0055-8

Automatic symbolic compositional verification by learning assumptions

Wonhong Nam · P. Madhusudan · Rajeev Alur

Published online: 17 May 2008

Abstract Compositional reasoning aims to improve scalability of verification tools by reducing the original verification task into subproblems. The simplification is typically based on assume-guarantee reasoning principles, and requires user guidance to identify appropriate assumptions for components. In this paper, we propose a fully automated approach to compositional reasoning that consists of automated decomposition using a hypergraph partitioning algorithm for balanced clustering of variables, and discovering assumptions using the L^* algorithm for active learning of regular languages. We present a symbolic implementation of the learning algorithm, and incorporate it in the model checker NuSMV. In some cases, our experiments demonstrate significant savings in the computational requirements of symbolic model checking.

Keywords Formal verification · Symbolic model checking · Compositional verification · Assume-guarantee reasoning · Regular language learning · Hypergraph partitioning

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-

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

W. Nam (✉)
Pennsylvania State University, University Park, PA, USA
e-mail: wnam@psu.edu

P. Madhusudan
University of Illinois, Urbana-Champaign, IL, USA
e-mail: madhu@cs.uiuc.edu

R. Alur
University of Pennsylvania, Philadelphia, PA, USA
e-mail: alur@cis.upenn.edu

guarantee based compositional reasoning to verify that a system S satisfies a requirement φ typically consists of the following three steps: (1) *System Decomposition*: partitioning the system S into components $M_1, \dots, M_n$, (2) *Assumption Discovery*: finding an environment assumption A_i for each component M_i, and (3) *Assumption Checking*: verifying that the assumptions A_i are appropriate for proving or disproving the satisfaction of φ by S.

In this paper, we develop a fully automated framework for symbolic compositional verification by automatic partitioning and learning assumptions. Although a modular description of a system can suggest a natural decomposition, we have several reasons for automatic partitioning. The description of a system, particularly when transformed from a different language to the input language of a model checker, is often monolithic. In addition, the partition provided in the original model may 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 hypergraph partitioning [24, 25]. Given a system S with a set of variables X and a desired number n of components, we decompose the set X into n disjoint subsets $X_1, \dots, X_n$ so that each set X_i contains approximately the same number of variables while keeping the number of communication variables small. Each such variable partition X_i corresponds to a component M_i that controls these variables.

After partitioning, the main problem of compositional reasoning is to identify appropriate assumptions for all the components so that the assumption checking phase will succeed, and one promising solution is based on learning [6, 8, 14, 17, 32]. If a component M_i communicates with its environment via a set IO_i of boolean variables, then the assumption A_i can be viewed as a language over the alphabet 2^{IO_i}. The assumption checking constraints impose a lower and an upper bound on this language; verifying that the environment assumption is adequate for proving the safety property gives an upper bound on which behaviors can be allowed in the assumption, and verifying that the assumption of one component is guaranteed by the other components gives a lower bound on what behaviors it must include. In this paper, we employ two assume-guarantee reasoning rules; one is for a system with two components, and the other is for an arbitrary number of components. We compute this assumption using the L^* algorithm for learning a regular language using membership and equivalence queries [7, 37]. The learning-based approach produces a DFA, and the number of queries made by the learner is only polynomial in the size of the output automaton. The membership query is to test whether a given trace belongs to the desired assumption. The equivalence query is to test whether the current assumptions are adequate for the assumption checking phase. Both queries are implemented by using symbolic model checking techniques.

While the standard L^* algorithm is designed to learn a particular language, and the desired assumptions A_i belong to a class of languages containing all languages that satisfy all requirements of the assume-guarantee rule, we show that our strategy based on the L^* algorithm learns one of the languages correctly. The learning-based approach to automatic generation of assumptions is appealing as it builds the assumption incrementally guided by the model-checking queries, and if it encounters an assumption that is a witness of the requirement φ, the algorithm will stop and use it to prove the property.

We present our implementations of the automated compositional reasoning using a symbolic model checker NuSmv [12]. In our context, the size of the alphabet itself grows exponentially with the number of communication variables. Consequently, we propose a symbolic implementation of the L^* algorithm where the required data structures for representing membership information and the assumption automata are maintained compactly using BDDs [11]. In addition, we have enhanced our implementation with several 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.

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 trade-offs: (1) how do the more general assume-guarantee rule and the heuristics impact the performance? (2) what is the impact of the number of components on the overall computational requirements? and (3) how does the integrated tool, *Automatic Symbolic Compositional Verifier* (ASCV), compare with NuSmv? It turns out that the general assume-guarantee rule and the heuristics work pretty well. No conclusions can be drawn regarding whether a 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 resulting from 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 compared with the non-compositional analysis performed by the original NuSmv.

Related work

Application of regular language learning to formal verification field has received little attention in the past. However, since the use of learning algorithms for automatic discovery of assumptions [8, 14] was first proposed in 2003, several papers have been reported in this subject.

Our application of the learning technique is related to the work of Cobleigh et al. [14] who use the L^* algorithm to automatically construct assumptions for compositional verification. In order to verify that the composition of two components M_1 and M_2 satisfies a safety requirement φ, the authors propose to learn the assumption A on inputs to M_1 such that $M_1 \| A$ satisfies φ and M_2 satisfies A [14].

Compared to these papers, we believe that our work makes several contributions. First, we present a symbolic implementation of the learning algorithm, and this is essential since the alphabet is exponential in the number of communication variables. Second, we address and explain explicitly how the L^* algorithm designed to learn an unknown, but fixed, language is adapted to learn *some* assumption from a class of correct assumption languages. Third, we demonstrate the benefits of the method by incorporating it in a state-of-the-art publicly available symbolic model checker. Finally, our work includes an automatic decomposition strategy, heuristic improvements in computing the conjecture assumptions, and experiments to study several trade-offs.

After the work of Cobleigh et al., the use of learning algorithms has been further developed by many researchers: Alur et al. use predicate abstraction and learning to synthesize (dynamic) interfaces for Java classes [5]; Sharygina et al. [38] consider 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; Cobleigh et al. [15] report several experiments to test whether assume-guarantee reasoning could provide an advantage over monolithic verification. The work of Vardhan et al. [41, 42] uses learning to compute the set of reachable states for verifying infinite-state systems, while Peled et al. [35] use learning for *black box checking*, that is, verifying properties of partially specified implementations. In addition, Nam and Alur [33] have applied this learning technique to a planning problem under partial observability. Recently, Gupta et al. [20] show how to construct, using a SAT solver, a k-state automaton with the guarantee that the minimal DFA for every language between the lower and upper bounds for the language we are hoping to learn has at least k states. Sinha and Clarke [39] propose a lazy approach to assumption learning, which avoids an explicit enumeration of the exponential alphabet set by using symbolic alphabet clustering and iterative counter-example driven localized partitioning.

Compositional reasoning using assume-guarantee rules has a long history in the formal verification literature [1, 2, 19, 21, 23, 29, 31, 34, 36, 40]. While such reasoning is supported by some tools (e.g. MOCHA [3]), the challenging task of finding the appropriate assumptions is typically left to the user and only a few attempts have been made to automate the assumption generation (in the work of Alur et al. [4], the authors present some heuristics for automatically constructing assumptions using game-theoretic techniques).

Organization

The rest of this paper is organized as follows. In Sect. 2, we lay out the notions and the problem we consider in this paper. Section 3 presents our first step, automatic decomposition. Then, we explain a learning algorithm we employ for learning appropriate assumptions in Sect. 4. Section 5 provides, for two assume-guarantee rules, our algorithms which learn appropriate assumptions or find a counter-example. In Sect. 6, we illustrate our symbolic implementations for the algorithms, and heuristic improvements. Section 7 shows our experiments to study the effectiveness of our algorithms. Finally, we give a few conclusions in Sect. 8.

2 Preliminaries

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

2.1 Symbolic transition systems

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

– X is a finite set of boolean *variables*.
– $\mathit{Init}(X)$ is an *initial predicate* over X.
– $T(X, X')$ is a *transition predicate* over $X \cup X'$ (X' represents a set of variables encoding the successor states).

A *state* q of the transition system 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, q' denotes a state over X' such that $q'(x') = q(x)$ for every $x \in X$. 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 every 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* we consider in this paper is to check whether $S \models \varphi$ holds.

2.2 Decomposition into modules

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

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

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. Given a module $M(X_M, X_M^I, X_M^O, \mathit{Init}_M, T_M)$, a *run* of M, similar to the run of a symbolic transition system, is a sequence $q_0q_1\cdots$ where every q_i is a state over $X_M \cup X_M^I$, such that $\mathit{Init}(q_0[X_M], q_0[X_M^I])$ holds, and for each $i \geq 0$, $T(q_i[X_M], q_i[X_M^I], q_{i+1}'[X_M'])$ holds. Now, we can extended the notion of $\models$ to modules as follows: $M \models \varphi$ if for every run $q_0q_1\cdots$ of M, $\varphi(q_i)$ holds for every $i \geq 0$. Given a set of modules $M_1, \ldots, M_n$, where each $M_i = (X_{M_i}, X_{M_i}^I, X_{M_i}^O, \mathit{Init}_{M_i}, T_{M_i})$, we can compose them if for every i and j $(j \neq i)$, X_{M_i} is disjoint from X_{M_j}. We denote this composition as $M_1\|\cdots\|M_n$, and the composition can be considered as another module $M(X_M, X_M^I, X_M^O, \mathit{Init}_M, T_M)$ as follows:

- $X_M = X_{M_1} \cup \cdots \cup X_{M_n}$.
- $X_M^I = \bigcup_i X_{M_i}^I \setminus \bigcup_i X_{M_i}^O$.
- $X_M^O = \bigcup_i X_{M_i}^O$.
- $\mathit{Init}_M(X_M, X_M^I) = \mathit{Init}_{M_1} \wedge \cdots \wedge \mathit{Init}_{M_n}$.
- $T_M(X_M, X_M^I, X_M') = T_{M_1} \wedge \cdots \wedge T_{M_n}$.

Given a variable $x \in X$ of a symbolic transition system $S(X, \mathit{Init}, T)$, $\mathit{Depend}(x)$ denotes a set of variables $y \in X$ such that the value of y depends on the value of x in Init or T; e.g. $y' := x$. We assume that $\mathit{Depend}(x)$ for each variable x is given, and in practice this can be easily obtained from Init and T of the transition system. Given a transition system $S(X, \mathit{Init}, T)$ and a set $Y \subseteq X$ of variables, we define a module $S[Y]$, which intuitively is the projection of system S on variables Y, as the tuple $(X_{S[Y]}, X_{S[Y]}^I, X_{S[Y]}^O, \mathit{Init}_{S[Y]}, T_{S[Y]})$ where:

- $X_{S[Y]} = Y$.
- $X_{S[Y]}^I = \{x \in X \setminus Y \mid \exists y \in Y.\ y \in \mathit{Depend}(x)\}$.
- $X_{S[Y]}^O = \{y \in Y \mid \exists x \in X \setminus Y.\ x \in \mathit{Depend}(y)\}$.
- $\mathit{Init}_{S[Y]}(X_{S[Y]}, X_{S[Y]}^I)$ is an initial predicate (for variables $y \in Y$) over $X_{S[Y]} \cup X_{S[Y]}^I$.
- $T_{S[Y]}(X_{S[Y]}, X_{S[Y]}^I, X_{S[Y]}')$ is a transition predicate (for variables $y \in Y$) over $X_{S[Y]} \cup X_{S[Y]}^I \cup X_{S[Y]}'$.

Now, we can decompose a transition system $S(X, \mathit{Init}, T)$ into modules $S[X_1], \ldots, S[X_n]$ by partitioning X into $X_1, \ldots, X_n$ where $X = \bigcup_i X_i$ and every X_i is disjoint from each other.

For a transition system $S(X, \mathit{Init}, T)$ decomposed into $S[X_1], \ldots, S[X_n]$ where each $S[X_i] = (X_{S[X_i]}, X_{S[X_i]}^I, X_{S[X_i]}^O, \mathit{Init}_{S[X_i]}, T_{S[X_i]})$, each run of S is obviously a run of $S[X_1]\|\cdots\|S[X_n]$ and each run of $S[X_1]\|\cdots\|S[X_n]$ is also a run of S, since $X = \bigcup_i X_i$, and Init and T of S are the conjunction of each $\mathit{Init}_{S[X_i]}$ and the conjunction of each $T_{S[X_i]}$, respectively. Finally, given $S(X, \mathit{Init}, T)$ and a partition of X into disjoint subsets $X_1, \ldots, X_n$, $S[X_1]\|\cdots\|S[X_n] \models \varphi$ iff $S \models \varphi$.

2.3 Assume-guarantee rules

For a given run $q_0q_1\cdots$ of a module $M(X_M, X_M^I, X_M^O, \mathit{Init}_M, T_M)$, the *trace* is a sequence $q_0[X_M^{IO}]q_1[X_M^{IO}]\cdots$. 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 X_M^{IO}). Given two modules M_1 and M_2 that have the same input and output variables, we say M_1 is a *refinement* of M_2, denoted $M_1 \sqsubseteq M_2$, if $L(M_1) \subseteq L(M_2)$. For a trace set L over a variable set X_M^{IO} and a safety property φ over X_M^{IO}, we can extended the notion of $\models$ to trace sets as following: $L \models \varphi$ if for every trace $q_0q_1\cdots$ 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 discuss two assume-guarantee rules to prove that a safety property φ holds for a composition of modules.

Simple rule This assume-guarantee rule is to prove that a composition of two modules, $M_1\|M_2$ satisfies a safety property φ over X^{IO} where $X^{IO} = X_{M_1}^{IO} \cup X_{M_2}^{IO}$. In the rule below, A is an assumption module $(X_A, X_A^I, X_A^O, \mathit{Init}_A, T_A)$ with $X_A^I = X_{M_2}^I$ and $X_A^O = X_{M_2}^O$.

$$\text{Rule-S:}\quad \frac{\begin{array}{ll} M_1\|A \models \varphi & \text{(Pr1-S)} \\ M_2 \sqsubseteq A & \text{(Pr2-S)} \end{array}}{M_1\|M_2 \models \varphi}$$

The rule says that if there exists (some) module A such that the composition of M_1 and A is safe (i.e. satisfies the property φ) and M_2 refines A, then $M_1||M_2$ satisfies φ. We can view such an A as an *appropriate assumption* between M_1 and M_2: it is an abstraction of M_2 (possibly admitting more behaviors than M_2) that is a strong enough assumption for M_1 to make in order to satisfy φ. Then, our aim with this rule is to find such an assumption A to show that $M_1\|M_2$ satisfies φ. A smaller assumption can save the more in terms of searching state space. This rule is sound and complete [34].

General rule While Rule-S can be applied to a composition of two modules, we need a more general rule to be used for a composition of an arbitrary number of modules, $M_1\|\cdots\|M_n$. The following assume-guarantee rule, Rule-G, can be used to prove that a composition of modules, $M_1\|\cdots\|M_n$ satisfies a safety property φ over X^{IO} $(=\bigcup_i X_{M_i}^{IO})$. In the rule below, let each A_i be a module such that $M_i||A_i$ is well defined and $X_{A_i}^I \cup X_{Ai}^O = X^{IO}$. Note that we are assuming that the safety property φ for this rule is a predicate over X^{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.

$$\text{Rule-G:}\quad \frac{\begin{array}{ll} M_1\|A_1 \models \varphi, \cdots, M_n\|A_n \models \varphi & \text{(Pr1-G)} \\ L^C(A_1)\|\cdots\|L^C(A_n) \models \varphi & \text{(Pr2-G)} \end{array}}{M_1\|\cdots\|M_n \models \varphi}$$

The rule above says that if there exist assumption modules $A_1, \ldots, A_n$ such that 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-G makes every assumption strong enough to make each M_i safe, and the second premise Pr2-G makes the assumptions weak enough to cover all the traces which can violate φ (i.e., for every trace violating φ, Pr2-G requires at least one assumption to contain it). This rule is sound and complete [8]. With this rule, our aim is to discover such assumptions $A_1, \ldots, A_n$ to show that $M_1 \| \cdots \| M_n$ satisfies φ. For $n = 2$, while Rule-S requires only an assumption A for M_2, Rule-G tries to find two assumptions A_1 and A_2 simultaneously.

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 $X_1, \ldots, X_n$, and to check $S[X_1] \| \cdots \| S[X_n] \models \varphi$ using the above assume-guarantee rules. The challenges of this model checking problem are (1) how to find a good variable partition and (2) how to find assumptions satisfying both the above premises.

3 Automatic partitioning

Our first step is automatic decomposition. This partitioning problem is, given a transition system $S(X, \mathit{Init}, T)$ and an integer $n \geq 2$, to decompose X into disjoint subsets $X_1, \ldots, X_n$. There exist approximately $n^{|X|}$ possible partitions. Among them, we want a partition that minimizes memory usage for assumption construction and commitment in our assume-guarantee reasoning. The memory usage, however, cannot be predicted precisely. Thus, we revise our goal to find a partition that has small number of variables required in each step of the assume-guarantee reasoning because the number of possible states is exponential in the number of variables. More precisely, the alternative goal is to find a partition that minimizes $max_i(|X_i \cup X^{IO}_{S[X_i]}|)$ where $X^{IO}_{S[X_i]}$ is the set of I/O variables of the module $S[X_i]$. This partitioning problem is NP-complete [25].

We reduce our problem to a well-known partitioning problem called the *hypergraph partitioning problem* which is also NP-complete. For the reduction, we relax our goal as following; given a transition system $S(X, \mathit{Init}, T)$, and an integer $n \geq 2$, the goal of automatic partitioning is to find a partition decomposing X into n disjoint subsets such that (1) the number variables per module is approximately the same (2) modules corresponding to each variable subset have as few input/output variables as possible.

A *hypergraph* $hG(V, hE)$ is defined as a set of vertices V and a set of *hyperedges* hE 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 edges are pairs of two vertices. Given a hypergraph $hG(V, hE)$ 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, $V_1, \ldots, V_k$ 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 reformulated as the k-way hypergraph partitioning problem. Given a transition system $S(X, \mathit{Init}, T)$, we construct a hypergraph $hG(V, hE)$ as follows. $V = \{v_x \mid x \in X\}$. For each $x \in X$, we have a hyperedge e_x that contains the corresponding vertex v_x and also vertices v_y such that $y \in \mathit{Depend}(x)$, meaning that the variable y has a read-dependency on x. Finally, hE is the set of all edges e_x. Then, after hypergraph partitioning, $V_1, \ldots, V_k$ corresponds to $X_1, \ldots, X_n$ 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 exists 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 $S[X_j]$ and also an output variable of $S[X_i]$. For the overall load imbalance tolerance c, a large value as c can reduce the number of I/O variables but it causes larger imbalance among modules in terms of their number of variables. On the other hand, a small value as c increases I/O variables. Therefore, we perform partitioning with six different values (e.g. 1.0, 1.2, ..., 2.0) and pick the partition that has the minimum value of $\max_i(|X_i \cup X^{IO}_{S[X_i]}|)$. While ‘1.0’ as c value requires that every partition has the same number of variables, ‘2.0’ means that each partition may have between twice and half the average value. We ignore more biased c values since we believe that the more biased partitions gain less benefit from compositional reasoning.

Many researchers have studied the hypergraph partitioning problem and developed tools, and among them we use hMETIS [24]. hMETIS is a state-of-the-art hypergraph partitioning tool 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 and refines 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 [27] and FM [16]. In addition, it is fast enough to produce high quality bisections of hypergraphs with 100,000 vertices in 3 minutes [25].

4 L^* algorithm

According to the literature on active learning[1] for regular languages, the active learning techniques for regular languages can be classified by the teachers they use: (1) a teacher answering membership and equivalence queries [7, 37], and (2) a teacher answering only equivalence queries but she always provides the lexicographically first counter-example [10, 22]. To the best of our knowledge, the best result of the first techniques requires $O(|\Sigma|n^2 + n \log m)$ membership queries and at most $n - 1$ equivalence queries, where n is the number of states in the target DFA and m is the length of the longest counter-example provided by the teacher [37]. On the other hand, the second one needs $O(|\Sigma|n^2)$ equivalence queries [10]. The total number of queries needed is similar in both techniques, but we believe that in our problem equivalence queries require more computation then membership queries. Thus, we employ the learning technique with a teacher for membership and equivalence queries, which requires fewer equivalence queries.

The L^* algorithm learns an unknown regular language and generates a minimal DFA accepting the language by asking membership and equivalence queries to a teacher. This algorithm had been introduced by Angluin [7], but later its efficiency was improved by Rivest and Schapire [37] (see [26] for a gentle introduction to these algorithms). In this paper, we employ the improved version. The algorithm infers the structure of the DFA by asking a teacher membership and equivalence queries.

Figure 1 illustrates the L^* algorithm [37]. Let U be the unknown regular language and Σ be its alphabet. At any given time, the L^* algorithm has, in order to construct a conjecture DFA, information about a finite collection of strings over Σ, classified either as members or non-members of U. This information is maintained in an *observation table* (R, E, G) where

[1]Note that while *passive learning* learns a target from given sets of *positive samples* and *negative samples*, in *active learning* a learner itself selects and asks queries to a teacher (oracle) who answers correctly for the queries, and learns the target based on the answers from the teacher. Typically, the former cannot learn the target exactly, but the latter can.

```
1:  $R := \{\varepsilon\}$;
2:  $E := \{\varepsilon\}$;
3:  $G[\varepsilon, \varepsilon] := member(\varepsilon \cdot \varepsilon)$;
4:  foreach $(a \in \Sigma)$ { $G[\varepsilon \cdot a, \varepsilon] := member(\varepsilon \cdot a \cdot \varepsilon)$; }
5:  repeat:
6:      while $((r_{new} := closed(R, E, G)) \neq null)$ {
7:          $add(R, r_{new})$;
8:          foreach $(a \in \Sigma), (e \in E)$ { $G[r_{new} \cdot a, e] := member(r_{new} \cdot a \cdot e)$; }
9:      }
10:     $C := makeConjectureDFA(R, E, G)$;
11:     if $((cex := equivalent(C)) = null)$ then return $C$;
12:     else {
13:         $e_{new} := findSuffix(cex)$;
14:         $add(E, e_{new})$;
15:         foreach $(r \in R), (a \in \Sigma)$ {
16:             $G[r, e_{new}] := member(r \cdot e_{new})$;
17:             $G[r \cdot a, e_{new}] := member(r \cdot a \cdot e_{new})$;
18:         }
19:     }
```

Fig. 1 L^* algorithm

R and E are sets of strings over Σ, and G is a function from $(R \cup R \cdot \Sigma) \times E$ to $\{0, 1\}$. More precisely, R is a set of representative strings for states in the conjecture DFA such that each representative string $r_q \in R$ for a state q leads from the initial state (uniquely) to the state q, and E is a set of experiment suffix strings that are used to distinguish states (i.e., for any two states q_1 and q_2 of the DFA, there is a string $e \in E$ such that $member(r_{q_1} \cdot e) \neq member(r_{q_2} \cdot e)$ where r_{q_1} and r_{q_2} are representative strings for q_1 and q_2, respectively). G maps strings $\sigma = \sigma_1 \cdot \sigma_2$ (where $\sigma_1 \in R \cup R \cdot \Sigma$ and $\sigma_2 \in E$) to 1 if σ is in U, and to 0 otherwise. Initially, R and E are set to $\{\varepsilon\}$, and G is initialized using membership queries for every string in $(R \cup R \cdot \Sigma) \times E$ (lines 3–4). In line 6, the algorithm checks whether the observation table is *closed*. The function *closed(R, E, G)* returns *null* (meaning true) if for every $r \in R$ and $a \in \Sigma$, there exists $r' \in R$ such that $G[r \cdot a, e] = G[r', e]$ for every $e \in E$; otherwise, it returns $r \cdot a$ such that there is no r' satisfying the above condition. If the table is not closed, such an $r \cdot a$ (e.g., denoted by r_{new} in line 7) is simply added to R as a new state (representative string). The algorithm again updates G with regard to $r \cdot a$ (line 8). Once the table is closed, it constructs a conjecture DFA $C = (Q, q_0, F, \delta)$ as follows (line 10): $Q = R$, $q_0 = \varepsilon$, $F = \{r \in R \mid G[r, \varepsilon] = 1\}$, and for every $r \in R$ and $a \in \Sigma$, $\delta(r, a) = r'$ such that $G[r \cdot a, e] = G[r', e]$ for every $e \in E$. Finally, if the answer for the equivalence query is 'yes', it returns the current conjecture machine C; otherwise, a counter-example $cex \in ((L(C) \setminus U) \cup (U \setminus L(C))$ is provided by the teacher. The algorithm analyzes the counter-example cex in order to find the longest suffix e_{new} of cex that witnesses a difference between U and $L(C)$ (line 13). Adding e_{new} to E reflects the difference in the next conjecture by splitting states in C. It then updates G with respect to e_{new} (lines 15–18).

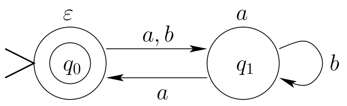
The L^* algorithm is guaranteed to construct a minimal DFA for the unknown regular language using only $O(|\Sigma|n^2 + n \log m)$ membership queries and at most $n - 1$ equivalence queries, where n is the number of states in the final DFA and m is the length of the longest counter-example provided by the teacher when answering equivalence queries. Moreover, the algorithm never constructs any intermediate automaton with more than n states.

G_1		E
		ε
R	ε	1
$R \cdot \Sigma$	a	0
	b	0

(a) G_1

G_2		E
		ε
R	ε	1
	a	0
$R \cdot \Sigma$	a	0
	b	0
	aa	1
	ab	0

(b) G_2



(c) C_1

Fig. 2 Observation tables and conjecture machines (part 1 of 3)

Example

Suppose that the unknown regular set U is the set of all strings over $\Sigma = \{a, b\}$ with an even number of a's and an even number of b's. We illustrate how the L^* algorithm in Fig. 1 constructs a minimal DFA representing U.

Initially, the algorithm sets R and E to $\{\varepsilon\}$, and asks membership queries for the strings ε, a and b in lines 3–4. The initial observation table G_1 is shown in Fig. 2a. This observation table is not closed, since there is no $r \in R$ such that $G_1[a, e] = G_1[r, e]$ for every $e \in E$. The L^* algorithm adds the string a to the set R in line 7 and then asks membership queries for the strings aa and ab to construct the observation table G_2 shown in Fig. 2b. This observation table is closed, and the algorithm constructs a conjecture C_1, shown in Fig. 2c. The initial state is q_0 and the accepting state is also q_0. In the following automata, a string labeled on each state is the representative string in R which is corresponding to the state. However, C_1 is not a correct DFA for U, so the teacher provides a counter-example. In this case, we assume that the counter-example is bb; $bb \in U \setminus L(C_1)$.

From the counter-example bb, the L^* algorithm adds the string b to the experimental string set E, which is returned by the function *findSuffix*() (lines 13–14). Then, it again updates the observation for the new experiment string (lines 15–18) and the result is G_3 in Fig. 3a. However, G_3 is not closed since there is no $r \in R$ such that $G_3[b, e] = G_3[r, e]$ for every $e \in E$. We add the string b to the set R and then ask membership queries for ba and bb. The result observation table is G_4 (Fig. 3b) which is now closed. The algorithm constructs a conjecture C_2 based on G_4, which is shown in Fig. 3c. However, C_2 still is not a correct DFA for U, so the teacher provides a counter-example. We assume that the teacher returns aba; $aba \in L(C_2) \setminus U$.

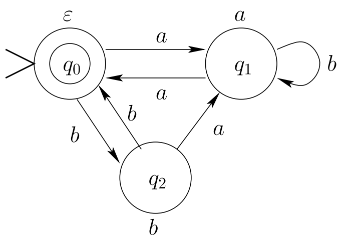
The function *findSuffix*() returns the string a as a new experiment string from the counter-example aba, and we add it to the set E. Then, the algorithm again updates the observation table for the new experiment string and the result is G_5 in Fig. 4a. Again, G_5 is not closed since there is no $r \in R$ such that $G_5[ab, e] = G_5[r, e]$ for every $e \in E$. We add the string ab to the set R and then ask membership queries for aba and abb to construct the observation table G_6 shown in Fig. 4b. This observation table is now closed, and the algorithm again constructs a conjecture C_3, shown in Fig. 4c. The conjecture DFA C_3 is a correct DFA for the language U and the teacher replies with 'yes' (line 11). Finally, the L^* algorithm terminates and returns C_3 as its output.

G_3		E	
		ε	b
R	ε	1	0
	a	0	0
$R\cdot\Sigma$	a	0	0
	b	0	1
	aa	1	0
	ab	0	0

(a) G_3

G_4		E	
		ε	b
R	ε	1	0
	a	0	0
	b	0	1
$R\cdot\Sigma$	a	0	0
	b	0	1
	aa	1	0
	ab	0	0
	ba	0	0
	bb	1	0

(a) G_4



(c) C_2

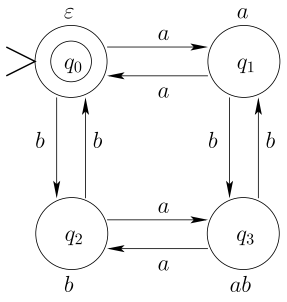
Fig. 3 Observation tables and conjecture machines (part 2 of 3)

G_5		E		
		ε	b	a
R	ε	1	0	0
	a	0	0	1
	b	0	1	0
$R\cdot\Sigma$	a	0	0	1
	b	0	1	0
	aa	1	0	0
	ab	0	0	0
	ba	0	0	0
	bb	1	0	0

(a) G_5

G_6		E		
		ε	b	a
R	ε	1	0	0
	a	0	0	1
	b	0	1	0
	ab	0	0	0
$R\cdot\Sigma$	a	0	0	1
	b	0	1	0
	aa	1	0	0
	ab	0	0	0
	ba	0	0	0
	bb	1	0	0
	aba	0	1	0
	abb	0	0	1

(b) G_6



(c) C_3

Fig. 4 Observation tables and conjecture machines (part 3 of 3)

5 Learning assumptions

In this section, for each of Rule-S and Rule-G in Sect. 2, we define the *weakest safe assumption* (*tuple*) which is a witness for the truth of a given invariant. We then establish that our verification algorithms based on the L^* algorithm converge to the weakest safe assumption (tuple) or, before that, conclude with a witness for the invariant.

5.1 Weakest safe assumption for Rule-S

Rule-S is a compositional rule to verify a transition system S consisting of two modules. To use this rule, we first partition S into two modules, $S[X_1]$ and $S[X_2]$ by automatic partitioning as explained in Sect. 3. Then, given two modules $S[X_1]$ and $S[X_2]$ and a safety property φ, in order to apply Rule-S, we need to find an assumption A that satisfies two premises, Pr1-S and Pr2-S of Rule-S. An assumption A is called a *safe assumption* if the assumption A satisfies Pr1-S (i.e. $S[X_1]\|A \models \varphi$), and an assumption A is called an *appropriate assumption* if the assumption A satisfies both of Pr1-S and Pr2-S. The *weakest safe assumption* W [18] is a module such that $S[X_1]\|W \models \varphi$ and $L(W) \supseteq L(A)$ for every safe assumption A (i.e. $S[X_1]\|A \models \varphi$). For a given module and a safety property, it is easy to see that W is guaranteed to exist, and is unique. Now, we show that the weakest safe assumption W is a witness for the truth of $S[X_1]\|S[X_2] \models \varphi$. That is, W gives proof whether $S[X_1]\|S[X_2] \models \varphi$ or not as an evidence.

Lemma 1 *If $S[X_1]\|S[X_2] \models \varphi$, then the weakest safe assumption W is an appropriate assumption with respect to* Rule-S.

Proof If $S[X_1]\|S[X_2]$ does indeed satisfy φ, then there exists an appropriate assumption A since Rule-S is complete. By definition, W satisfies Pr1-S. For the above appropriate assumption A, since $S[X_2] \sqsubseteq A$ and $L(A) \subseteq L(W)$, $S[X_2] \sqsubseteq W$ (Pr2-S). Hence, the weakest safe assumption W is an appropriate assumption, and witnesses that $S[X_1]\|S[X_2] \models \varphi$ through Rule-S. □

Lemma 2 *If $S[X_1]\|S[X_2] \not\models \varphi$, then the weakest safe assumption W is not an appropriate assumption with respect to* Rule-S.

Proof If $S[X_1]\|S[X_2]$ does not satisfy φ, then by the soundness of Rule-S, there is no appropriate assumption; hence W is not an appropriate assumption. Since W satisfies Pr1-S (by definition), W does not satisfy Pr2-S; i.e. there exists $\tau \in L(S[X_2]) \setminus L(W)$ that does not satisfy φ. □

The weakest safe assumption W can be represented as a DFA with the alphabet Q^{IO} (where Q^{IO} is a set of all states over X^{IO}) because $S[X_1]$ and $S[X_2]$ are finite. Therefore, we can learn the weakest safe assumption W which a witness for truth of $S[X_1]\|S[X_2] \models \varphi$, using the L^* algorithm for learning regular languages.

5.2 Weakest safe assumption tuple for Rule-G

Rule-G is a compositional rule to verify a transition system S consisting of an arbitrary number of modules. To use this rule, we decompose S into n modules. Then our aim is, given a set of modules $S[X_1], \dots, S[X_n]$ and a safety property φ, to verify that $S[X_1]\| \cdots \|S[X_n] \models \varphi$ by finding assumption modules $A_1, \dots, A_n$ that satisfy both premises, Pr1-G and Pr2-G of Rule-G. A tuple $(A_1, \dots, A_n)$ of assumptions is called a *Safe assumption Tuple* (ST) if the assumptions $A_1, \dots, A_n$ satisfy Pr1-G, and a tuple $(A_1, \dots, A_n)$ of assumptions is called an *Appropriate assumption Tuple* (AT) if the assumptions $A_1, \dots, A_n$ satisfy both of Pr1-G and Pr2-G. For every $S[X_i]$, the *weakest safe assumption W_i* is a module such that $S[X_i]\|W_i \models \varphi$ and $L(W_i) \supseteq L(A_i)$ for every A_i such that $S[X_i]\|A_i \models \varphi$. We denote such a tuple $(W_1, \dots, W_n)$ as the *Weakest safe assumption Tuple* (WT). As the weakest safe assumption for Rule-G, the WT is guaranteed to exist, and is unique. Now, we show that the WT is a witness for the truth of $S[X_1]\| \cdots \|S[X_n] \models \varphi$.

Lemma 3 *If $S[X_1]\| \cdots \|S[X_n] \models \varphi$, then the weakest safe assumption tuple $(W_1, \dots, W_n)$ is an appropriate assumption tuple with respect to* Rule-G.

Proof If $S[X_1]\| \cdots \|S[X_n]$ does indeed satisfy φ, then there exists an AT $(A_1, \dots, A_n)$ since the composition rule is complete. By definition, WT $(W_1, \dots, W_n)$ satisfies Pr1-G. For the above AT $(A_1, \dots, A_n)$, since for every i, $L^C(W_i) \subseteq L^C(A_i)$ and $L^C(A_1)\| \cdots \|L^C(A_n) \models \varphi$, it follows that $L^C(W_1)\| \cdots \|L^C(W_n) \models \varphi$ (Pr2-G). Hence WT $(W_1, \dots, W_n)$ is an appropriate assumption tuple that proves $S[X_1]\| \cdots \|S[X_n] \models \varphi$. □

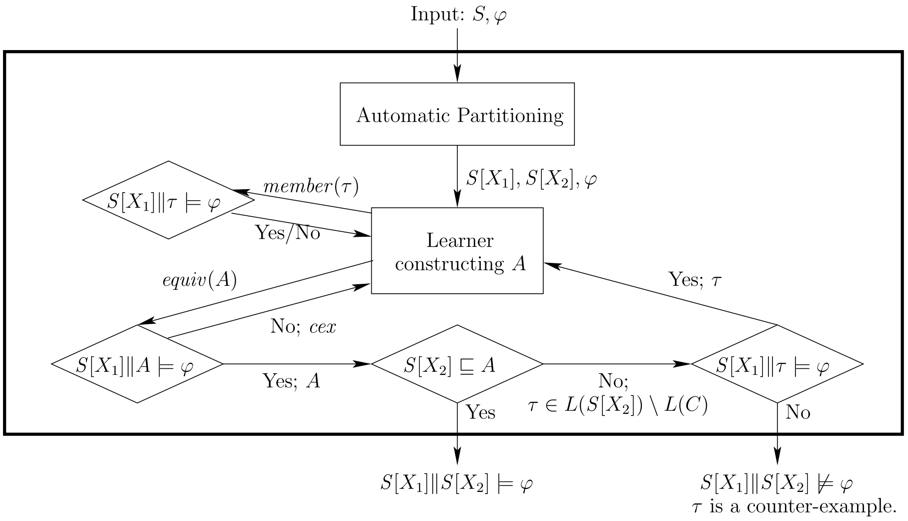

Input: S, φ
Automatic Partitioning
$S[X_1], S[X_2], \varphi$
$S[X_1]\|\tau \models \varphi$
$member(\tau)$
Yes/No
Learner constructing A
$equiv(A)$
Yes; τ
No; cex
$S[X_1]\|A \models \varphi$
Yes; A
$S[X_2] \sqsubseteq A$
No; $\tau \in L(S[X_2]) \setminus L(C)$
$S[X_1]\|\tau \models \varphi$
Yes
No
$S[X_1]\|S[X_2] \models \varphi$
$S[X_1]\|S[X_2] \not\models \varphi$
τ is a counter-example.

Fig. 5 Overview of compositional verification for Rule-S

Lemma 4 *If* $S[X_1]\| \cdots \|S[X_n] \not\models \varphi$, *then the weakest safe assumption tuple is not an appropriate assumption tuple with respect to* Rule-G.

Proof If $S[X_1]\| \cdots \|S[X_n]$ does not satisfy φ, then, by soundness of Rule-G, there is no appropriate assumption tuple, and hence the weakest assumption tuple cannot be appropriate. In fact, since WT $(W_1, \ldots, W_n)$ satisfies Pr1-G by definition, there exists $\tau \in L^C(W_1)\| \cdots \|L^C(W_n)$ violating φ. □

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

5.3 Compositional verification algorithm for Rule-S

Now, we present our verification algorithm for Rule-S. Given a transition system $S(X, \mathit{Init}, T)$ and an invariant property φ, ASCV_S algorithm (*Automatic Symbolic Compositional Verification* for Rule-S) decomposes X into two disjoint subsets, X_1 and X_2, and then checks $S[X_1]\|S[X_2] \models \varphi$ by learning the weakest safe assumption W which is a witness for the truth of the invariant.

Figure 5 illustrates the high-level overview of our compositional verification procedure. It first partitions S into two modules and the learner learns W by asking queries to the teacher. Given a string $\tau \in (Q^{IO})^*$, the teacher answers whether there is an execution of $S[X_1]$ consistent with τ, which violates φ; that is, whether $\tau \notin L(W)$. The learner constructs a conjecture assumption A for the module $S[X_1]$, 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-S and Pr2-S of the assume-guarantee rule. If a given assumption tuple satisfies both premises, we conclude $S[X_1]\|S[X_2] \models \varphi$; otherwise, the teacher produces a counter-example. More precisely, the teacher checking Pr1-S answers, given an

```
     Boolean ASCV_S($S, \varphi$)
1:   $M[\,]$ := AutomaticPartitioning($S$, 2);
2:   $A$ := InitializeAssumptions($M[\,], \varphi$);
3:   repeat:
4:       while(($cex$ := SafeAssumption($M[1], A, \varphi$)) $\neq$ null){
5:           UpdateAssumption($M[1]$, $A$, $cex$);
6:       }
7:       if(($cex$ := Refinement($M[2], A$)) = null) then {
8:           return true;
9:       } else {
10:          if(SafeTrace($M[1]$, $cex$)) then UpdateAssumption($M[1]$, $A$, $cex$);
11:          else return false;
12:      }
```

Fig. 6 ASCV_S algorithm

assumption A for a module $S[X_1]$, whether $S[X_1]\|A \models \varphi$; if not, it returns $cex \in L(A)$ violating φ (i.e. $cex \in L(A) \setminus L(W)$). The teacher for Pr2-S checks, given an assumption A, whether $S[X_2] \sqsubseteq A$; if not, it returns $\tau \in L(S[X_2]) \setminus L(A)$. For Pr1-S queries, *cex* is immediately used to update A, but for Pr2-S queries, we need an additional analysis. We check if the trace τ (which is a behavior of $S[X_2]$ but not in $L(A)$) keeps $S[X_1]$ safe. If it does not, we conclude that $S[X_1]\|S[X_2]$ does not satisfy φ; otherwise, we give τ back to the L^* algorithm as a counter-example.

One of the nice properties of the L^* algorithm is that it needs a polynomial number of queries in the size of the minimal automaton accepting the language learned (and polynomial in the lengths of the counter-examples provided by the teacher). Let us now explain which language our algorithm converges to and derive bounds on the size of the automaton, and simultaneously show that our procedure always terminates.

Note that all membership queries and all counter-examples provided by the teacher in our algorithm are consistent with respect to W (for equivalence queries, all the counter-examples are checked with W). Therefore, our algorithm eventually converges to the weakest safe assumption W that is a witness for the model-checking question. However, there can be early termination with a counter-example or an appropriate assumption satisfying both premises. Hence our procedure always halts and reports correctly whether $S[X_1]\|S[X_2]$ satisfies φ, and in doing so, it never generates any assumption with more states than the minimal DFA accepting W.

Figure 6 illustrates the ASCV_S algorithm. Given a transition system S and a safety property φ, ASCV_S algorithm first decomposes S into two modules and assigns them to $M[1]$ and $M[2]$ (line 1), and it constructs the initial conjecture assumption A according to the rule of the L^* algorithm (line 2). Then, we repeat asking two sub-queries for equivalence and updating the current conjecture assumption A; if either of two sub-queries returns a counter-example *cex*, the algorithm updates A using *cex* (lines 4–12). Updating the conjecture assumption involves asking membership queries. In more detail, we first check that the current assumption A is a safe assumption such that $M[1]\|A \models \varphi$ by a function `Safe-Assumption()`. If so, we have A satisfying Pr1-S; otherwise we update A with respect to *cex* (line 5). Once we have A satisfying Pr1-S, the algorithm checks Pr2-S by a function `Refinement()`. If the function returns *null*, then we conclude $S \models \varphi$ since the current assumption A satisfies both premises; otherwise, we are provided a counter-example *cex*. Lines 10–11 analyze whether *cex* is a real counter-example for the invariant; if *cex* indeed violates φ for $M[1]$, then we conclude $S \not\models \varphi$. Otherwise, it is a spurious counter-example and we again update A using *cex*.

Summarizing the above argument, we have:

Theorem 1 *The algorithm depicted in Fig. 6, when input a system S and a safety property φ, always halts, and returns* true *iff $S \models \varphi$. Moreover, if S is split into M_1 and M_2, then the algorithm never constructs an assumption automaton A that has more states than the minimal deterministic automaton accepting the weakest safe assumption W with respect to (M_1, M_2).*

5.4 Compositional verification algorithm for Rule-G

In this section, we present our verification algorithm for the general rule Rule-G. Given a transition system $S(X, \mathit{Init}, T)$, an invariant property φ, and an integer $n \geq 2$, ASCV_G algorithm (*Automatic Symbolic Compositional Verification* for Rule-G) decomposes X into n disjoint subsets $X_1, \dots, X_n$ and then checks $S[X_1] \| \cdots \| S[X_n] \models \varphi$ by learning the WT (Weakest safe assumption Tuple) which is a witness for the truth of the invariant. For learning the WT, ASCV_G algorithm provides teachers who answer membership queries and equivalence queries, which correspond with the WT $(W_1, \dots, W_n)$. Our algorithm is trying to learn a set of languages for $W_1, \dots, W_n$, but this can be viewed as n copies of L^* learning in parallel.

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

In the ASCV_G algorithm, since all answers from teachers are always consistent with the WT (for equivalence queries, counter-examples are checked with each W_i), ASCV_G algorithm will converge to the WT which a witness for the truth of $S \models \varphi$, in 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 7 illustrates the ASCV_G algorithm. Given a transition system S, a safety property φ, and an integer n, ASCV_G algorithm first decomposes S into n modules and assigns them to an array $M[\,]$ (line 1), and it constructs the initial conjecture machines $A[i]$ according to the rule of the L^* algorithm (line 2). Then, we repeat asking two sub-queries for equivalence and updating the current conjecture machines; if either of the sub-queries returns a counter-example *cex*, the algorithm updates the current conjecture machines using *cex*

```
     Boolean ASCV_G($S, \varphi, n$)
 1:  $M[\,]$ := AutomaticPartitioning($S, n$);
 2:  $A[\,]$ := InitializeAssumptions($M[\,], \varphi$);
 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:      }
 9:      if(($cex$ := DischargeAssumptions($A[1], \cdots, A[n], \varphi$)) = $null$){
10:          return $true$;
11:      } else {
12:          $IsRealCex$ := $true$;
13:          foreach($1 \le i \le n$) {
14:              if(SafeTrace($M[i], cex$) {
15:                  UpdateAssumption($M[i], A[i], cex$);
16:                  $IsRealCex$ := $false$;
17:              }
18:          }
19:          if($IsRealCex$) return $false$;
20:      }
```

Fig. 7 ASCV_G algorithm

(lines 4–20). In more detail, we check that for every i, the current conjecture $A[i]$ is a safe assumption such that $M[i] \| A[i] \models \varphi$ by a function `SafeAssumption()`. If so, we have $A[1], \ldots, A[n]$ satisfying Pr1-G; otherwise (i.e., for some i, we have a counter-example *cex* violating φ), we update $A[i]$ with respect to *cex* (line 6). Once we have $A[1], \ldots, A[n]$ satisfying Pr1-G, the algorithm checks Pr2-G by a function `DischargeAssumptions()`. If the function returns *null*, then we conclude $S \models \varphi$ since $A[1], \ldots, A[n]$ satisfy both premises; otherwise, we are provided a counter-example *cex*. Lines 12–19 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 $A[i]$ which is the current conjecture for $M[i]$ not violating φ.

Summarizing the above argument, we have:

Theorem 2 *The algorithm depicted in Fig. 7, when input a system S and a safety property φ, always halts, and returns true iff $S \models \varphi$. Moreover, if S is split into $M_1, \ldots, M_n$, then the algorithm constructs assumption automata $A_1, \ldots, A_n$, where no A_i has ever more states than that of the minimal deterministic automaton for W_i, where $(W_1, \ldots, W_n)$ is the weakest safe assumption tuple for $(M_1, \ldots, M_n)$.*

6 Symbolic implementations

ASCV_S algorithm and ASCV_G algorithm can be implemented implicitly as well as explicitly. However, as input/output variables increase, the number of alphabet symbols of the languages we want to learn increases exponentially. In explicit implementations, 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. Therefore, we introduce a symbolic implementation for learning-based compositional verification (first introduced in [6, 32]).

6.1 Implementation of the ASCV_S algorithm

6.1.1 Data structures

For symbolic implementation of the two algorithms, ASCV_S and ASCV_G, we already defined a symbolic transition system and decomposition to modules implicitly in Sect. 2. For describing a symbolic implementation for the L^* algorithm, we first explain the essential data structures the L^* algorithm needs, and then present implicit data structures corresponding to them, which we use in our implementation. The L^* algorithm uses the following data structures for an observation table:

– `string R[]`: each `R[i]` is a representative string for i-th state q_i in the conjectured DFA.
– `string E[]`: each `E[i]` is i-th experiment string.
– `boolean G1[][]`: each `G1[i][j]` is the result of the membership query for `R[i]`$\cdot$`E[j]`.
– `boolean G2[][][]`: each `G2[i][j][k]` is the result of the membership query for `R[i]`$\cdot a_j \cdot$`E[k]` where a_j is the j-th alphabet symbol in Σ.

Note that $G : (R \cup R\cdot\Sigma) \times E \to \{0, 1\}$ of the observation table is split into two arrays, `G1` and `G2`; `G1` is an array for the function from $R \times E$ to $\{0, 1\}$ and `G2` is for a function from $R \times \Sigma \times E$ to $\{0, 1\}$. The L^* algorithm initializes the data structures as following: `R[0]=E[0]=`ε, `G1[0][0]=`$\mathit{member}(\varepsilon \cdot \varepsilon)$, and `G2[0][i][0]=`$\mathit{member}(\varepsilon \cdot a_i \cdot \varepsilon)$ (for every $a_i \in \Sigma$). When introducing a new state or a new experiment string, the L^* algorithm adds to `R[]` or `E[]`, and updates `G1` and `G2` by membership queries. These arrays also represent the edges of the conjecture machine: there is an edge from state q_i to q_j on a_k when `G2[i][k][l]=G1[j][l]` for every `l`.

For symbolic implementations of ASCV_S and ASCV_G algorithm, we do not wish to construct conjecture DFAs by explicit membership queries since $|\Sigma|$ is too large. To identify an alphabet symbol labeled on each edge, the explicit L^* algorithm asks, for each state r, alphabet symbol a and experiment e, if $r \cdot a \cdot e$ is a member. However, in our symbolic technique, given a source state r and the *boolean vector* v of a target state, we compute the set of alphabet symbols a's such that for every $j \le |v|$, $\mathit{member}(r \cdot a \cdot e_j) = v[j]$. The edges from a state r to the target state captured by the boolean vector v, are labeled by the representation of the set of symbols that are possible on these edges. This symbolic method can reduce the cost of each explicit membership query. Now, we have the following data structures for symbolic ASCV_S algorithm (for ASCV_G algorithm, we have one set of the following data structures for each module):

– `int nQ`: the number of states in the current DFA.
– `int nE`: the number of experiment strings.
– `BDD R[]`: each `R[i]` ($0 \le$ `i` $<$ `nQ`) is a BDD over X_1 to represent the set of states of the module $S[X_1]$ that are reachable from an initial state of $S[X_1]$ by the representative string r_i of the i-th state q_i: $\mathit{postImage}(\mathit{Init}_{S[X_1]}, r_i)$.
– `BDD E[]`: each `E[i]` ($0 \le$ `i` $<$ `nE`) is a BDD over X_1 to capture a set of states of $S[X_1]$ from which some state violating φ is reachable by the i-th experiment string e_i: $\mathit{preImage}(\neg\varphi(X_1), e_i)$.

- `booleanVector G1[]`: each `G1[i]` ($0 \leq$ `i` $<$ `nQ`) is the *boolean vector* for the state q_i, where the length of each boolean vector always equals to `nE`. Note that as `nE` is increased, the length of each boolean vector is also increased. For $i \neq j$, `G1[i]` $\neq$ `G1[j]`. Each element `G1[i][j]` of `G1[i]` ($0 \leq$ `j` $<$ `nE`) indicates whether `R[i]` and `E[j]` have empty intersection, and this holds precisely when $r_i \cdot e_j$ is a member where r_i is a representative string for `R[i]` and e_j is an experiment string for `E[j]`.
- `booleanVector Cd[]`: every iteration of the L^* algorithm splits some states of the current conjecture DFA by a new experiment string. More precisely, the new experiment splits every state into two *state candidates*, and among them, only reachable candidates are constructed as states of the next conjecture DFA. The `Cd[]` vector encodes all these state candidates and each element is the boolean vector of each candidate. $|\texttt{Cd}| = 2\cdot\texttt{nQ}$.

Given $S = S[X_1] \| S[X_2]$ and φ, we initialize the data structures as follows. `R[0]` is the BDD for $Init_{S[X_1]}$ and `E[0]` is the BDD for $\neg\varphi$ since the corresponding representative and experiment string are ε, and `G1[0][0] = 1` since we assume that every initial state satisfies φ.

We also represent the conjecture assumption implicitly as $A(X_A, X^{IO}, Init_A, F_A, T_A)$ with the following components:

- X_A is a set of boolean variables representing states in A ($|X_A| = \lceil \log_2 \texttt{nQ} \rceil$) Valuations of the variables can be encoded from the index for `R`.
- X^{IO} is a set of boolean variables defining its alphabet, which comes from $S[X_1]$ and $S[X_2]$.
- $Init_A(X_A)$ is an initial state predicate over X_A. $Init_A(X_A)$ is encoded from the index of the state q_0: $Init_A(X_A) = \bigwedge_{x \in X_A}(x \equiv 0)$.
- $F_A(X_A)$ is a predicate for accepting states. It is encoded from the indices of the states q_i such that `G1[`i`][0]=1`.
- $T_A(X_A, X^{IO}, X'_A)$ is a transition predicate over $X_A \cup X^{IO} \cup X'_A$; that is, if $T_A(i, a, j) = true$, then the DFA has an edge from state q_i to q_j labeled by a.

This symbolic DFA $A(X_A, X^{IO}, Init_A, F_A, T_A)$ can be easily converted to a module $M_A(X_A, X^I, X^O, Init_A, T_A)$.

6.1.2 Subfunctions

We implement, by using the above symbolic data structures, every function in ASCV_S algorithm as follows. Figure 8 also illustrates the pseudo-codes for the important ones.

- `BDD[] InitializeAssumptions(Module[] M, Property` φ`)`: given a module array and a safety property, this function initializes an assumption module $A[i]$ for each module $M[i]$ based on the L^* algorithm. For initialization, it asks implicit membership queries.
- `BDD[] SafeAssumption(Module M, SymbolicDFA A, Property` φ`)`: given a module M, a symbolic assumption DFA $A(X_A, X^{IO}, Init_A, F_A, T_A)$ and a property φ, this function checks if $M \| M_A \models (F_A \rightarrow \varphi)$ holds using symbolic forward reachability analysis, where M_A is a symbolic module converted from A. If so, it returns *null*; otherwise, it returns a BDD array as a counter-example.
- `void UpdateAssumption(Module M, SymbolicDFA A, BDD[] cex)`: given a module M, an assumption A and a counterexample *cex*, it updates the current assumption A using *cex*. The function first computes the longest suffix of *cex* which shows a difference between A and the weakest safe assumption for M, and then it adds the suffix to the experiment string set E. This addition introduces new states and edges, and requires membership queries.

```
BDD[] SafeAssumption(Module M, SymbolicDFA A, Property $\varphi$){
    BDD $\upsilon := false$;
    BDD $\tau := Init_M(X_M) \wedge Init_A(X_A)$;
    while($\tau \neq false$){
        if(($\tau \wedge F_A \wedge \neg\varphi) \neq false$) then return $computeCex(\tau \wedge F_A \wedge \neg\varphi)$;
        else{
            $\upsilon := \upsilon \vee \tau$;
            $\tau := (\exists X_M, X^{IO}, X_A.\ \tau \wedge T_M \wedge T_A)[X'_M \rightarrow X_M, X'_A \rightarrow X_A]$;
            $\tau := \tau \wedge \neg\upsilon$;
    } }
    return null;
}

BDD[] Refinement(Module M, SymbolicDFA A){
    BDD $\upsilon := false$;
    BDD $\tau := Init_M(X_M) \wedge Init_A(X_A)$;
    while($\tau \neq false$){
        if(($\tau \wedge \neg F_A) \neq false$) then return $computeCex(\tau \wedge \neg F_A)$;
        else{
            $\upsilon := \upsilon \vee \tau$;
            $\tau := (\exists X_M, X^{IO}, X_A.\ \tau \wedge T_M \wedge T_A)[X'_M \rightarrow X_M, X'_A \rightarrow X_A]$;
            $\tau := \tau \wedge \neg\upsilon$;
    } }
    return null;
}

void UpdateAssumption(Module M, SymbolicDFA A, BDD[] cex){
     addE(findSuffix($cex$));
     foreach ($0 \leq i <$ nQ) {
         foreach ($0 \leq j < 2\cdot$nQ) {
             $eds :=$ edges($i$, Cd[$j$]);
             if ($eds \neq false$) then {
                 if ($\neg$isInR(Cd[$j$])) then addR($i$, $eds$, Cd[$j$]);
                 $C := addEdges(i, eds, indexofR($Cd[$j$]))$;
     } } }
}
```

Fig. 8 Implementation of subfunctions for ASCV_S algorithm

- `BDD[] Refinement(Module M, SymbolicDFA A)`: given a module M and a symbolic assumption DFA A, it checks whether $M \sqsubseteq A$ holds. The return value is similar with `SafeAssumption()`. Since A is again a symbolic DFA, we can simply implement it by symbolic reachability computation for the product of M and M_A. If it reaches the non-accepting state of A, the sequence reaching the non-accepting state is a witness showing $M \not\sqsubseteq A$.
- `boolean SafeTrace(Module M, BDD[]` τ`)`: given a module M and a trace τ represented by a BDD array, it executes M according to τ. If the execution reaches a state violating φ, it returns *false*; otherwise, returns *true*.

The function, `UpdateAssumption()` invokes several subfunctions as follows. Figure 9 also illustrates the pseudo-code for some of them.

- `BDD edges(Module M, int i, booleanVector v)`: given a module M, an integer i and a boolean vector v ($0 \leq i <$ `nQ`, $|v| =$ `nE`), this function returns a BDD over X^{IO} representing the set of alphabet symbols for the module M, by which there is an edge from the state q_i to a state that has v as its boolean vector.
- `void addR(Module M, int i, BDD eds, booleanVector v)`: when we introduce a new state for a module (its predecessor state is q_i, the BDD representing edges from q_i is *eds* and its boolean vector is v), `addR(M, i, eds, v)` adds the new state to `R` and updates `G1` and `nQ` for the module M.
- `void addE(Module M, BDD[] exp)`: given a module M and a new experiment string represented as an array of BDDs (where each BDD of the array encodes the corresponding state in the experiment string), this function updates `E, G1` and `nE` for the module M. It also constructs a new set `Cd[]` of state candidates for the next iteration.
- `boolean isInR(Module M, booleanVector v)`: given a module M and a boolean vector v, it checks, for the module M, whether `v` = `G1[i]` for some `i`.
- `BDD[] findSuffix(Module M, BDD[] cex)`: given a module M and a counterexample *cex* (from equivalence queries) represented by a BDD array, this function returns a BDD array representing the longest suffix of *cex* which witnesses the difference between the conjecture DFA and the weakest safe assumption W for the module M. From the shortest prefix of *cex*, we first identify the representative string for the prefix by traversing the current conjecture machine, and replace the prefix part of *cex* with the representative string. We then check whether this string is safe. If so, the rest of the string is the suffix that is able to distinguish between the conjecture and the weakest safe assumption. Otherwise, we try again this checking with the next shortest prefix.

6.2 Implementation of the ASCV_G algorithm

6.2.1 *Data structures and subfunctions*

The implementation for ASCV_G algorithm shares most of data structures with ASCV_S implementation. The main difference is that as ASCV_G algorithm needs one assumption for each module, we have n copies of data structures for the observation tables and assumption machines.

For subfunctions, ASCV_G implementation invokes, instead of `Refinement()`, the following function `DischargeAssumptions()` as the second premise Pr2-G in Rule-G is different with Pr2-S. For other subfunctions in ASCV_G algorithm, the implementation is the same as in the ASCV_S algorithm.

- `BDD[] DischargeAssumptions(SymbolicDFA[] A, Property` φ`)`: it checks the second premise Pr2-G: $L^C(A_1) \| \cdots \| L^C(A_n) \models \varphi$. For every A_i, we first construct a module encoding the complement DFA of A_i. This complementation 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.

```
BDD edges(Module M, int i, booleanVector v){
    BDD eds := $true$; // eds is a BDD over $X^{IO}$.
    foreach $(0 \le j <$ nE$_M$){ // In the below, $X_M^L = X_M \setminus X^{IO}$.
        if (v[$j$]) then
            eds := eds $\wedge \neg(\exists X_M^L, X_M{}'.$ R$_M$[i]$(X_M) \wedge T_M(X_M, X_M^I, X'_M) \wedge$ E$_M$[j]$(X'_M))$;
        else eds := eds $\wedge (\exists X_M^L, X_M{}'.$ R$_M$[i]$(X_M) \wedge T_M(X_M, X_M^I, X'_M) \wedge$ E$_M$[j]$(X'_M))$;
    }
    return eds;
}

void addR(Module M, int i, BDD eds, booleanVector v){
    BDD io := $pickOne$(eds); // io is a BDD representing one alphabet symbol.
    R$_M$[nQ] := $(\exists X_M, X_M^I.$ R$_M$[i]$(X_M) \wedge$ io $\wedge T_M(X_M, X_M^I, X'_M))[X'_M \to X_M]$;
    G1$_M$[nQ$_M$++] := v;
}

void addE(Module M, BDD[] exp){
    BDD b := $\varphi$; // b is a BDD over $X_M$.
    for ($j :=$ $length$(exp); $j > 0$; $j$--)
        b := $\exists X_M^I, X'_M.$ b$(X'_M) \wedge$ exp[$j$] $\wedge T_M(X_M, X_M^I, X'_M)$;
    E$_M$[nE$_M$] := ¬b;
    foreach $(0 \le i <$ nQ$_M$) {
        if ((R$_M$[i] $\wedge$ E$_M$[nE$_M$]) = $false$) then G1$_M$[i][nE$_M$] := 1;
        else G1$_M$[i][nE$_M$] := 0;
        foreach $(0 \le j <$ nE$_M$) {
            Cd$_M$[2i][j] := G1$_M$[i][j];
            Cd$_M$[2i+1][j] := G1$_M$[i][j];
        }
        Cd$_M$[2i][nE$_M$] := 0;
        Cd$_M$[2i+1][nE$_M$] := 1;
    }
    nE$_M$++;
}

BDD[] findSuffix(Module M, BDD[] cex){
    BDD [] prefix := $\varepsilon$;
    BDD [] suffix := cex;
    repeat:
        prefix := prefix · $head$(suffix); // concatenation
        suffix := $tail$(suffix);
        rep := $getRepresentativeString$(prefix);
        if SafeTrace(M, rep · suffix) then return suffix;
}
```

Fig. 9 Subfunctions for `UpdateAssumption()`

6.2.2 Early falsification

In previous implementations of learning-based compositional verification [6, 14, 17], we have found a possible optimization that allows us to conclude earlier $S \not\models \varphi$ with a counter-example. In Fig. 7, if the trace *cex* acquired from the function `DischargeAssumptions()` reaches some state violating φ for every $S[X_i]$, then we conclude that the invariant is false (line 19). That is, in the case that the invariant is indeed false, the algorithm cannot finish 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 may be a counter-examples for $S \models \varphi$. In our implementation for ASCV_G, if *cex* obtained from `SafeAssumption()` is a feasible trace for every other module $S[X_j](j \neq i)$, then we declare *cex* as a counter-example for $S \not\models \varphi$. Otherwise (*cex* violates φ in $S[X_i]$, but it is infeasible for some other module), we update the current assumption for $S[X_i]$ to rule out *cex* as in the previous algorithms. We believe that the additional feasibility checking adds little to the computational overhead, but can sometimes falsify the invariant earlier. We will present examples where the algorithm is able to terminate much earlier than experiments without *early falsification* in Sect. 7. 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 ASCV_G algorithm (see Fig. 7).

```
5:      while(($cex :=$ SafeAssumption($M[i], A[i], \varphi$)) $\neq null$){
5′:         if(EarlyFalsify($cex, i$)) return $false$;
6:          UpdateAssumption($M[i], A[i], cex$);
7:      }
```

6.2.3 Edge deletion for safe assumptions

The ultimate goal of our model-checking problem is to quickly discover a small appropriate assumption (tuple) or a counter-example for $S \models \varphi$. ASCV_S and ASCV_G algorithm, however, only guarantee that we can eventually learn the weakest safe assumption (tuple) whose size is, in theory, exponential in the size of each module in the worst case. That is, both algorithms based on the L^* algorithm may keep introducing new states for conjecture machines until converging on a very large weakest safe assumption (tuple), even though there may exist smaller appropriate assumption (tuple) than the weakest safe assumption (tuple). We have experienced many cases where our algorithm needs many iterations to converge on the WT (lines 5–6 in Fig. 7). The optimal solution for this problem is to learn the smallest appropriate assumption (tuple) in terms of the number of states rather than the weakest safe assumption (tuple), but this is a computationally hard problem.

Instead, we propose, for ASCV_G algorithm, a simple heuristic called *edge deletion* where we retry, without introducing new states, to check Pr1-G and Pr2-G after eliminating some edges from the current assumption. More precisely, when we are given a counter-example *cex* from `SafeAssumption(`$M[i], A_i, \varphi$`)`, *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 $M[i]\|A_i$, and we can find 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 `Safe-Assumption()` 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 can encounter a smaller AT than the original algorithm. Section 7 shows evidence of this benefit.

7 Experiments

We have implemented our automatic symbolic compositional verification algorithms with the BDD package in the symbolic model checker NuSMV [12]. For experiments, we have six sets of examples where five sets are collected from the NuSMV package and one is artificial. The primary selection criterion was to include examples for which NuSMV takes a long time or fails to complete. All experiments have been performed on a Sun-Blade-1000 workstation using a 750 MHz UltraSPARC III processor, 1 GB memory and SunOS 5.10. We first explain the artificial example (called '`simple`') to illustrate our method and then report results on '`simple`' and five example sets from the NuSMV package.

Example: *simple* Module $S[X_1]$ has a variable x (initially set to 0 and updated by the rule $x' := y$ in each round where y is an input variable) and a variable array that does not affect x at all. Module $S[X_2]$ has a variable y (initially set to 0 and is never updated) and also a variable array that does not affect y at all. For $S[X_1]\|S[X_2]$, we want to check that x is always 0. Both arrays in $S[X_1]$ and $S[X_2]$ are from an example *swap* known to be hard for BDD encoding [30]. Our tool explores $S[X_1]$ and $S[X_2]$ separately with a two-state assumption (which allows only $y = 0$), while ordinary model checkers will search whole state space of $S[X_1]\|S[X_2]$. In our experiment, we have four instances, `simple8`, `simple9`, `simple10` and `simple11`, with arrays which have 8 to 11 4-bit elements, respectively.

The second example set, `guidance`, is a model of a space shuttle digital autopilot. We added redundant variables to the original model and did not use a given variable ordering information as both tools finished fast with the original model and the ordering. The specifications were picked from the given pool: `guidance_sp1`, `guidance_sp2` and `guidance_sp3` have the same models but have different specifications. The third set, `barrel`, is an example for bounded model checking and no variable ordering works well for BDD-based tools. `barrel_sp1` has an invariant derived from the original, but `barrel_sp2` has our own ones. `barrel_sp1` and `barrel_sp2` have the same model scaled-up from the original, but with different initial predicates.

The fourth set, `msi`, is a MSI cache protocol model and shows how the tools scale on a real example. We scaled-up the original model with 3 nodes: `msi3` has 3 nodes, `msi4` has 4 nodes and `msi5` has 5 nodes. They have the same specification. `robot_sp1` and `robot_sp2` are robotics controller models and we again added redundant variables to the original model, as in the case of `guidance` example. The last set is `syncarb` that is the

Table 1 Comparison between ASCV_S and ASCV_G

Example	Spec	tv	ASCV_S				ASCV_G					
name			mx	*IO*	Time	Peak BDD	np	mx	*IO*	Time	Peak BDD	F/D
simple8	True	69	37	4	19.2	607,068	2	36	4	4.9	605,024	D
simple9		78	42	5	106	828,842	2	41	5	31.3	620,354	D
simple10		86	46	5	754	3,668,980	2	46	5	223	2,218,762	D
simple11		94	50	5	4601	12,450,004	2	50	5	1527	9,747,836	D
guidance_sp1	False	135	118	23	124	686,784	2	89	18	–	–	–
guidance_sp2	True	122	105	22	196	1,052,660	4	59	18	6.6	359,744	D
guidance_sp3	True	122	93	46	357	619,332	2	76	15	–	–	–
barrel_sp1	False	60	35	10	20.3	345,436	2	35	10	–	–	–
barrel_sp2	True	60	35	10	23.4	472,164	2	35	10	–	–	–
msi3	True	45	37	25	2.1	289,226	2	37	19	0.3	50,078	D
msi4		57	49	25	37.0	619,332	2	49	22	1.8	524,286	D
msi5		70	62	26	1183	6,991,502	2	60	25	31.9	2,179,926	D
robot_sp1	False	92	89	12	1271	4,169,760	2	52	5	283	1,905,008	F
robot_sp2	True	92	75	12	1604	2,804,368	2	50	7	9.5	427,196	D

model of a synchronous arbiter. We have two instances with 6 and 7 elements, and for each instance there are two specifications which are from the given pool.

We compare ASCV_S (*Automatic Symbolic Compositional Verifier* for Rule-S) with ASCV_G (*Automatic Symbolic Compositional Verifier* for Rule-G). We then present effects of the number of partitions and our heuristics (early falsification and edge deletion in Sect. 6.2). Finally, we compare our ASCV_G with the invariant checking (with early termination) to NUSMV 2.3.0. Each result table has the number of variables in total (tv), I/O variables, $\max_i(|X_i \cup X^{IO}_{S[X_i]}|)$ (mx), execution time in seconds, the peak BDD size and the number of states in the assumptions we learn (asm). Entries denoted '–' mean that a tool did not finish within 2 hours. For ASCV_G, columns denoted 'F/D' mean that early falsification or the edge deletion contributes to concluding earlier, and 'np' stands for the number of partitions.

ASCV_S vs. ASCV_G Compared with ASCV_S, the ASCV_G has the following additional features: a symmetric compositional rule, early falsification and edge deletion. Table 1 presents that ASCV_G shows better performance in 10 of the 14 examples in terms of required time and memory. In more detail, for the examples where ASCV_G shows better performance, it terminates 3–30 times faster than ASCV_S. On the other hand, it cannot finish for the rest. Since we have selected the examples in Table 1 from [6], for which ASCV_S terminated within the time limit, there is no example where ASCV_S does not finish. For memory usage, while in `simple8` ASCV_G reduces the peak BDD size by a small amount, it consumes memory 3–5 times less than ASCV_S does for `guidnace_sp2`, `msi3` and `robot_sp2`. We omit the results for `syncarb` since both tools fail to learn small assumptions.

The number of partitions We have experimented with our example sets by using ASCV_G to explain how increasing the number of partitions affects the performance. However, some of them show that the large number of partitions helps our tool to terminate earlier, but in others it does adversely. Table 2 illustrates selected examples which show this phenomenon. In

Table 2 Effect of the number of partitions

np	simple11						
	Spec	tv	mx	*IO*	Time	Peak BDD	asm
2			49	5	1526	9,747,836	2,2
3	True	94	61	37	1.8	497,714	2,2,2
4			53	37	0.7	217,686	2,2,2,2
np	guidance_sp2						
	Spec	tv	mx	*IO*	Time	Peak BDD	asm
2			82	18	1680	612,178	2,2
3	True	122	61	23	34	614,222	2,2,2
4			59	33	6.6	359,744	2,2,2,2
np	robot_sp1						
	Spec	tv	mx	*IO*	Time	Peak BDD	asm
2			52	5	283	1,905,008	3,2
3	False	92	62	30	–	–	Too many
4			64	46	–	–	Too many
np	syncarb7_sp2						
	Spec	tv	mx	*IO*	Time	Peak BDD	asm
2			21	21	332	7,700,770	131,131
3	True	21	21	21	643	14,870,100	35,19,35
4			21	21	4520	31,234,364	11,11,19,19

simple11 and guidance_sp2, increasing the number of partitions saves significantly in terms of time and BDD usage by discovering small assumptions. However, in robot_sp1 and syncarb7_sp2, it needs more time and memory due to large assumptions. The partitioning result influences the appropriate assumption set, but we do not know which set is better and which assumption our learning algorithm eventually learns among the set. Hence, it seems to be a hard problem to decide which partitioning is best.

Early falsification & edge deletion In Table 3, we present how our new features of ASCV_G allow it to terminate earlier. It shows only interesting examples where ASCV_G, with or without early falsification and edge deletion, can terminate in the time limit. If the given invariant holds, the edge deletion heuristic can save the number of states of assumptions in many examples. If the invariant does not hold, early falsification seems to be useful. They, however, may waste effort unless they can succeed, since they require extra computation. For the simple set, edge deletion helps ASCV_G find appropriate assumptions twice faster in terms of running time, and for msi it provides a significant saving where it allows verification to finish. In syncarb set, however, these features affect performance adversely by wasting extra computation.

ASCV_G vs. NuSMV Finally, Table 4 presents the comparison between ASCV_G and NuSMV. We present examples for which at least one of tools terminates and NuSMV takes a long time or fails to complete, as well as syncarb7_sp1 and syncarb7_sp2 that explains a possible reason why our technique may show worse performance than NuSMV. In 10 examples, ASCV_G is significantly better than NuSMV where we have found small assumptions. That is, in simple8, simple9, msi3, msi4, and robot_sp2, ASCV_G

Table 3 With/without early falsification and edge deletion

Example name	Spec	tv	np	mx	*IO* var	Without F/D Time	Without F/D Peak BDD	Without F/D asm	With F/D Time	With F/D Peak BDD	With F/D asm	With F/D F/D
simple8		69	2	36	4	10.3	605,024	2,3	4.9	605,024	2,2	D
simple9	True	78	2	41	5	58.3	624,442	2,3	31.3	620,354	2,2	D
simple10		86	2	45	5	441	2,997,526	3,2	223	1,849,462	2,2	D
simple11		94	2	49	5	3044	9,747,836	2,3	1526	9,747,836	2,2	D
guidance_sp2	True	122	2	105	18	1634	1,066,968	2,37	1603	612,178	2,2	D
msi3		45	2	37	19	–	–	–	0.3	49,056	2,2	D
msi4	True	57	2	49	22	–	–	–	1.8	524,286	2,2	D
msi5		70	2	60	25	–	–	–	31.9	2,179,926	2,2	D
robot_sp1	False	92	2	52	5	529	2,275,994	3,58	283	1,905,008	3,2	F
robot_sp2	True	92	2	50	7	10.4	529,396	2,3	9.5	427,196	2,2	D
syncarb6_sp1	False	18	2	18	18	28.2	1,384,810	67,67	125	1,536,066	67,67	–
syncarb6_sp2	True	18	2	18	18	30.4	1,274,434	67,67	86.6	1,280,566	67,67	–
syncarb7_sp1	False	21	2	21	21	351	9,948,148	131,131	957	10,512,054	131,131	–
syncarb7_sp2	True	21	2	21	21	332	7,700,770	131,131	720	8,182,126	131,131	–

Table 4 Comparison between ASCV_G and NUSMV

Example name	Spec	tv	ASCV_G np	ASCV_G mx	ASCV_G *IO*	ASCV_G Time	ASCV_G Peak BDD	NUSMV Time	NUSMV Peak BDD
simple8		69	2	36	4	4.9	605,024	269	3,993,976
simple9	True	78	2	41	5	31.3	620,354	4032	32,934,972
simple10		86	4	50	37	1.0	330,106	–	–
simple11		94	4	53	37	0.7	217,686	–	–
guidance_sp2	True	122	4	59	18	6.6	359,744	–	–
msi3		45	2	37	19	0.3	50,078	157	1,554,462
msi4	True	57	2	49	22	1.8	524,286	3324	16,183,370
msi5		70	2	60	25	31.9	2,179,926	–	–
robot_sp1	False	92	2	52	5	283	1,905,008	654	2,729,762
robot_sp2	True	92	2	50	7	9.5	427,196	1039	1,117,046
syncarb7_sp1	False	21	2	21	21	351	9,948,148	0.1	5,110
syncarb7_sp2	True	21	2	21	21	332	7,700,770	0.1	3,066
barrel_sp1	False	60	–	–	–	–	–	1201	28,118,286
barrel_sp2	True	60	–	–	–	–	–	4886	36,521,170

reduces the required time or memory by two or three orders of magnitude. In addition, in `simple10`, `simple11`, `guidance_sp2`, and `msi5`, it converts infeasible problems into feasible ones. In `syncarb7_sp1` and `syncarb7_sp2`, however, the assumptions we have learned are relatively large (with 131 states for each) and we believe that the large size of assumptions is a main reason of negative results in these examples. Also, it can explain why ASCV_G cannot complete within the timeout in `barrel_sp1` and `barrel_sp2`, since ASCV_G constructed assumption candidates with a huge number of states until timeout.

8 Conclusions

For scalability of algorithmic verification, we have proposed a solution that combines symbolic state-space exploration, compositional reasoning via an assume-guarantee rule, computational learning for generating assumptions, and automatic decomposition. Given a state-transition system and a safety property, we check that the system satisfies the property as an invariant using assume-guarantee reasoning. In the proposed technique, we first decompose the system into small sub-modules, and then learn an environment assumption for each sub-module which is adequate for proving or disproving the satisfaction of the property. Our experiments demonstrate promising savings in terms of time and memory usage.

There are many directions for future work. First, we want to study heuristics to discover much smaller assumptions, although finding the smallest assumption is computationally hard. Second, we plan to consider circular assume-guarantee rules as well as other non-circular rules to identify more appropriate rules for the learning technique. Finally, it is worth pointing out that, while our implementations use BDD-based state-space exploration, the approach can easily be adopted to permit other model checking strategies such as SAT-based model checking [9, 30] and counter-example guided abstraction refinement [13, 28].

References

1. Abadi M, Lamport L (1995) Conjoining specifications. ACM Trans Program Lang Syst (TOPLAS) 17:507–534
2. Alur R, Henzinger T (1999) Reactive modules. Form Methods Syst Des 15(1):7–48. Invited submission to FLoC'96 special issue. A preliminary version appears in *Proceedings of the 11th LICS, 1996*
3. Alur R, Henzinger T, Mang F, Qadeer S, Rajamani S, Tasiran S (1998) MOCHA: Modularity in model checking. In: Proceedings of the 10th international conference on computer aided verification, pp 516–520
4. Alur R, de Alfaro L, Henzinger T, Mang F (1999) Automating modular verification. In: CONCUR'99: Concurrency theory, tenth international conference. LNCS, vol 1664. Springer, Berlin, pp 82–97
5. Alur R, Cerný P, Madhusudan P, Nam W (2005) Synthesis of interface specifications for Java classes. In: Proceedings of the 32nd symposium on principles of programming languages, POPL 2005, pp 98–109
6. Alur R, Madhusudan P, Nam W (2005) Symbolic compositional verification by learning assumptions. In: Proceedings of the 17th international conference of computer aided verification, CAV 2005, pp 548–562
7. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75:87–106
8. Barringer H, Pasareanu C, Giannakopoulou D (2003) Proof rules for automated compositional verification through learning. In: Proceedings of the 2nd international workshop on specification and verification of component based systems
9. Biere A, Cimatti A, Clarke E, Zhu Y (1999) Symbolic model checking without BDDs. In: Proceedings of the 5th international conference on tools and algorithms for the construction and analysis of systems, pp 193–207
10. Birkendorf A, Böker A, Simon H-U (2000) Learning deterministic finite automata from smallest counterexamples. SIAM J Discrete Math 13(4):465–491
11. Bryant R (1986) Graph-based algorithms for boolean-function manipulation. IEEE Trans Comput, C-35(8)
12. Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A (2002) NuSMV Version 2: An OpenSource tool for symbolic model checking. In: Proceedings of the 14th international conference on computer-aided verification (CAV 2002). LNCS, vol 2404. Springer, Berlin, pp 359–364
13. Clarke E, Grumberg O, Jha S, Lu Y, Veith H (2000) Counterexample-guided abstraction refinement. In: Proceedings of international conference on computer aided verification (CAV'00), pp 154–169
14. Cobleigh J, Giannakopoulou D, Pasareanu C (2003) Learning assumptions for compositional verification. In: Proceedings of the 9th international conference on tools and algorithms for the construction and analysis of software. LNCS, vol 2619. Springer, Berlin, pp 331–346
15. Cobleigh J, Avrunin G, Clarke L (2006) Breaking up is hard to do: An investigation of decomposition for assume-guarantee reasoning. In: Proceedings of the international symposium on software testing and analysis, pp 97–108

16. Fiduccia C, Mattheyses R (1982) A linear-time heuristic for improving network partitions. In: Proceedings of the 19th design automation conference, pp 241–247
17. Giannakopoulou D, Pasareanu C (2005) Learning-based assume-guarantee verification. In: Proceeding of the 12th international spin workshop, pp 282–287
18. Giannakopoulou D, Pasareanu C, Barringer H (2002) Assumption generation for software component verification. In: Proceedings of 17th IEEE international conference on automated software engineering (ASE 2002), pp 3–12
19. Grümberg O, Long D (1994) Model checking and modular verification. ACM Trans Program Lang Syst 16(3):843–871
20. Gupta A, McMillan K, Fu Z (2007) Automated assumption generation for compositional verification. In: Proceedings of the 19th international conference of computer aided verification, CAV 2007, pp 420–432
21. Henzinger T, Qadeer S, Rajamani S (1998) You assume, we guarantee: Methodology and case studies. In: CAV 98: Computer-aided verification. LNCS, vol 1427. Springer, Berlin, pp 521–525
22. Ibarra O, Jiang T (1991) Learning regular languages from counterexamples. J Comput Syst Sci 43(2):299–316
23. Jones C (1981) Development methods for computer programs including a notion of interference. PhD thesis, Oxford University
24. Karypis G, Kumar V (1999) Multilevel *k*-way hypergraph partitioning. In: Proceedings of the 36th conference on design automation, pp 343–348
25. Karypis G, Aggarwal R, Kumar V, Shekhar S (1999) Multilevel hypergraph partitioning: applications in VLSI domain. IEEE Trans Very Large Scale Integr (VLSI) Syst 7(1):69–79
26. Kearns M, Vazirani U (1994) An introduction to computational learning theory. MIT Press, Cambridge
27. Kernighan B, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49(2):291–307
28. Kurshan R (1994) Computer-aided Verification of Coordinating Processes: The automata-theoretic approach. Princeton University Press, Princeton
29. McMillan K (1997) A compositional rule for hardware design refinement. In: Proceedings of the 9th international conference on computer aided verification, pp 24–35
30. McMillan K (2002) Applying SAT methods in unbounded symbolic model checking. In: Proceedings of the 14th international conference on computer aided verification. LNCS, vol 2404. Springer, Berlin, pp 250–264
31. Misra J, Chandy K (1981) Proofs of networks of processes. IEEE Trans Softw Eng 7(4):417–426
32. Nam W, Alur R (2006) Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In: Proceedings of the 4th international symposium on automated technology for verification and analysis (ATVA'06), pp 170–185
33. Nam W, Alur R (2007) Learning plans for safety and reachability goals with partial observability. Technical Report MS-CIS-07-16, University of Pennsylvania
34. Namjoshi K, Trefler R (2000) On the completeness of compositional reasoning. In: Proceedings of the 12th international conference of computer aided verification, CAV 2000, pp 139–153
35. Peled D, Vardi M, Yannakakis M (2002) Black box checking. J Autom Lang Comb 7(2):225–246
36. Pnueli A (1984) In transition from global to modular temporal reasoning about programs. In: Logics and Models of Concurrent Systems. Springer, New York, pp 123–144
37. Rivest R, Schapire R (1993) Inference of finite automata using homing sequences. Inf Comput 103(2):299–347
38. Sharygina N, Chaki S, Clarke E, Sinha N (2005) Dynamic component substitutability analysis. In: Proceedings of the international symposium of formal methods Europe, pp 512–528
39. Sinha N, Clarke E (2007) SAT-based compositional verification using lazy learning. In: Proceedings of the 19th international conference of computer aided verification, CAV 2007, pp 39–54
40. Stark E (1985) A proof technique for rely-guarantee properties. In: FST & TCS 85: Foundations of software technology and theoretical computer science. LNCS, vol 206. Springer, Berlin, pp 369–391
41. Vardhan A, Viswanathan M (2006) Lever: A tool for learning based verification. In: Proceedings of 18th international conference on computer aided verification (CAV 2006), pp 471–474
42. Vardhan A, Sen K, Viswanathan M, Agha G (2004) Actively learning to verify safety properties for FIFO automata. In: Proceedings of 24th foundations of software technology and theoretical computer science. LNCS, vol 3328. Springer, Berlin, pp 494–505