Symbolic Compositional Verification by Learning Assumptions *

Rajeev Alur[1], P. Madhusudan[2], and Wonhong Nam[1]

[1] University of Pennsylvania
[2] University of Illinois at Urbana-Champaign
alur@cis.upenn.edu, madhu@cs.uiuc.edu, wnam@cis.upenn.edu

Abstract. The verification problem for a system consisting of components can be decomposed into simpler subproblems for the components using assume-guarantee reasoning. However, such compositional reasoning requires user guidance to identify appropriate assumptions for components. In this paper, we propose an automated solution for discovering assumptions based on 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. Our experiments demonstrate significant savings in the computational requirements of symbolic model checking.

1 Introduction

In spite of impressive progress in heuristics for searching the reachable state-space of system models, scalability still remains a challenge. Compositional verification techniques address this challenge by a "divide and conquer" strategy aimed at exploiting the modular structure naturally present in system designs. One such prominent technique is the *assume-guarantee* rule: to verify that a state property φ is an invariant of a system M composed of two modules M_1 and M_2, it suffices to find an abstract module A such that (1) the composition of M_1 and A satisfies the invariant φ, and (2) the module M_2 is a refinement of A. Here, A can be viewed as an assumption on the environment of M_1 for it to satisfy the property φ. If we can find such an assumption A that is significantly smaller than M_2, then we can verify the requirements (1) and (2) using automated search techniques without having to explore M. In this paper, we propose an approach to find the desired assumption A automatically in the context of symbolic state-space exploration.

If M_1 communicates with M_2 via a set X of common boolean variables, then the assumption A can be viewed as a language over the alphabet 2^X. We compute this assumption using the L^* algorithm for learning a regular language using membership and equivalence queries [6, 21]. The learning-based approach produces a *minimal* DFA, and the number of queries is only polynomial in

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

the size of the output automaton. The membership query is to test whether a given sequence σ over the communication variables belongs to the desired assumption. We implement this as a symbolic invariant verification query that checks whether the module M_1 composed with the sequence σ satisfies φ [16]. For an equivalence query, given a current conjecture assumption A, we first test whether M_1 composed with A satisfies φ using symbolic state-space exploration. If not, the counter-example provided by the model checker is used by the learning algorithm to revise A. Otherwise, we test if M_2 refines A, which is feasible since A is represented as a DFA. If the refinement test succeeds, we can conclude that M satisfies the invariant, otherwise the model checker gives a sequence σ allowed by M_2, but ruled out by A. We then check if the module M_1 stays safe when executed according to σ: if so, σ is used as a counter-example by the learning algorithm to adjust A, and otherwise, σ is a witness to the fact that the original model M does not satisfy φ.

While the standard L^* algorithm is designed to learn a particular language, and the desired assumption A belongs to a class of languages containing all languages that satisfy the two requirements of the assume-guarantee rule, we show that the above strategy works 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 has a small representation as a minimal DFA, the algorithm will stop and use it to prove the property. 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 automaton are maintained compactly using ordered BDDs [9] for processing the communication variables.

For evaluating the proposed approach, we modified the state-of-the-art symbolic model checker NuSMV [10]. In Section 5, we report on a few examples where the original models contain around 100 variables, and the computational requirements of NuSMV are significant. The only manual step in the current prototype involves specifying the syntactic decomposition of the model M into modules M_1 and M_2. While the proposed compositional approach does not always lead to improvement (this can happen when no "good" assumption exists for the chosen decomposition into modules M_1 and M_2), dramatic gains are observed in some cases reducing either the required time or memory by one or two orders of magnitude, or converting infeasible problems into feasible ones.

Finally, it is worth pointing out that, while our prototype uses BDD-based state-space exploration, the approach can easily be adopted to permit other model checking strategies such as SAT-based model checking [8, 18] and counter-example guided abstraction refinement [15, 11].

Related Work Compositional reasoning using assume-guarantee rules has a long history in the formal verification literature [22, 13, 1, 4, 17, 14, 19]. While such reasoning is supported by some tools (e.g. MOCHA [5]), 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 [3], the authors present some heuristics for automatically constructing assumptions using game-theoretic techniques).

Our work is inspired by the recent series of papers by the researchers at NASA Ames on compositional verification using learning [12, 7]. Compared to these papers, we believe that our work makes three 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. Finally, we demonstrate the benefits of the method by incorporating it in a state-of-the-art publicly available symbolic model checker.

It is worth noting that recently the L^* algorithm has found applications in formal verification besides automating assume-guarantee reasoning: our software verification project JIST uses predicate abstraction and learning to synthesize (dynamic) interfaces for Java classes [2]; [23] uses learning to compute the set of reachable states for verifying infinite-state systems; while [20] uses learning for *black box checking*, that is, verifying properties of partially specified implementations.

2 Symbolic modules

In this section, we formalize the notion of a symbolic module, the notion of composition of modules and explain the assume-guarantee rule we use in this paper.

Symbolic modules In the following, for any set of variables X, we will denote the set of primed variables of X as $X' = \{x' \mid x \in X\}$. A predicate φ over X is a boolean formula over X, and for a valuation s for variables in X, we write $\varphi(s)$ to mean that s satisfies the formula φ.

A *symbolic module* is a tuple $M(X, X^I, X^O, Init, T)$ with the following components:

- X is a finite set of boolean *variables* controlled by the module,
- X^I is a finite set of boolean *input variables* that the module reads from its environment; X^I is disjoint from X,
- $X^O \subseteq X$ is a finite set of boolean *output variables* that are observable to the environment of M,
- $Init(X)$ is an *initial state predicate* over X,
- $T(X, X^I, X')$ is a *transition predicate* over $X \cup X^I \cup X'$ where X' represents the variables encoding the successor state.

Let $X^{IO} = X^I \cup X^O$ denote the set of communication variables. A *state* s of M is a valuation of the variables in X; i.e. $s : X \to \{true, false\}$. Let S denote the set of all states of M. An *input state* s^I is a valuation of the input variables

X^I and an *output state* s^O is a valuation of X^O. Let S^I and S^O denote the set of input states and output states, respectively. Also, $S^{IO} = S^I \times S^O$. For a state s over a set X of variables, let $s[Y]$, where $Y \subseteq X$ denote the valuation over Y obtained by restricting s to Y.

The semantics of a module is defined in terms of the set of runs it exhibits. A *run* of M is a sequence $s_0, s_1, \cdots$, where each s_i is a state over $X \cup X^I$, such that $Init(s_0[X])$ holds, and for every $i \geq 0$, $T(s_i[X], s_i[X^I], s'_{i+1}[X'])$ holds (where $s'_{i+1}(x') = s_{i+1}(x)$, for every $x \in X$). For a module $M(X, X^I, X^O, Init, T)$ and a *safety property* $\varphi(X^{IO})$, which is a boolean formula over X^{IO}, we define $M \models \varphi$ if, for every *run* $s_0, s_1, \cdots$, for every $i \geq 0$, $\varphi(s_i)$ holds. Given a *run* $s_0, s_1, \cdots$ of M, the *trace* of M is a sequence $s_0[X^{IO}], s_1[X^{IO}], \cdots$ of input and output states. Let us denote the set of all the traces of M as $L(M)$. Given two modules $M_1 = (X_1, X^I, X^O, Init_1, T_1)$ and $M_2 = (X_2, X^I, X^O, Init_2, T_2)$ 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 $L(M_1) \subseteq L(M_2)$.

Composition of modules The synchronous composition operator $\|$ is a commutative and associative operator that composes modules. Given two modules $M_1 = (X_1, X_1^I, X_1^O, Init_1, T_1)$ and $M_2 = (X_2, X_2^I, X_2^O, Init_2, T_2)$, with $X_1 \cap X_2 = \emptyset$, $M_1 \| M_2 = (X, X^I, X^O, Init, T)$ is a module where:

- $X = X_1 \cup X_2$, $X^I = (X_1^I \cup X_2^I) \setminus (X_1^O \uplus X_2^O)$, $X^O = X_1^O \uplus X_2^O$,
- $Init(X) = Init_1(X_1) \wedge Init_2(X_2)$,
- $T(X, X^I, X') = T_1(X_1, X_1^I, X_1') \wedge T_2(X_2, X_2^I, X_2')$.

We can now define the model-checking problem we consider in this paper:

> Given modules $M_1 = (X_1, X_1^I, X_1^O, Init_1, T_1)$ and $M_2 = (X_2, X_2^I, X_2^O, Init_2, T_2)$, with $X_1 \cap X_2 = \emptyset$, $X_1^I = X_2^O$ and $X_1^O = X_2^I$ (let $X^{IO} = X_1^{IO} = X_2^{IO}$), and a safety property $\varphi(X^{IO})$, does $(M_1 \| M_2) \models \varphi$?

Note that we are assuming that the safety property φ is a predicate over the common communication variables X^{IO}. This is not a restriction: to check a property that refers to private variables of the modules, we can simply declare them to be outputs.

Assume-guarantee rule We use the following assume-guarantee rule to prove that a safety property φ holds for a module $M = M_1 \| M_2$. In the rule below, A is a module that has the same input and output variables as M_2:

$$\frac{M_1 \| A \models \varphi \qquad M_2 \sqsubseteq A}{M_1 \| M_2 \models \varphi}$$

The rule above 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 *adequate 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. Our aim is to construct such an assumption A to show that $M_1 \| M_2$ satisfies φ. This rule is sound and complete [19].

3 Assumption Generation via Computational Learning

Given a symbolic module $M = M_1 \| M_2$ consisting of two sub-modules and a safety property φ, our aim is to verify that M satisfies φ by finding an A that satisfies the premises of the assume-guarantee rule explained in Section 2. Let us fix a pair of such modules $M_1 = (X_1, X_1^I, X_1^O, Init_1, T_1)$ and $M_2 = (X_2, X_2^I, X_2^O, Init_2, T_2)$ for the rest of this section.

Let L_1 be the set of *all* traces $\rho = s_0, s_1, \cdots$, where each $s_i \in S^{IO}$, such that either $\rho \notin L(M_1)$ or $\varphi(s_i)$ holds for all $i \geq 0$. Thus, L_1 is the largest language for M_1's environment that can keep M_1 safe. Note that the languages of the candidates for A that satisfy the first premise of the proof rule is precisely the set of all subsets of L_1.

Let L_2 be the set of traces of M_2, that is, $L(M_2)$. The languages of candidates for A that satisfy the second premise of the proof rule is precisely the set of all supersets of L_2. Since M_1 and M_2 are finite, it is easy to see that L_1 and L_2 are in fact regular languages. Let B_1 be the module corresponding to the minimum state DFA accepting L_1.

The problem of finding A satisfying both proof premises hence reduces to checking for a language which is a superset of L_2 and a subset of L_1. To discover such an assumption A, our strategy is to construct A using a *learning algorithm for regular languages*, called the L^* algorithm. The L^* algorithm is an algorithm for a learner trying to learn a *fixed* unknown regular language U through membership queries and equivalence queries. Membership queries ask whether a given string is in U. An equivalence query asks whether a given language $L(C)$ (presented as a DFA C) equals U; if so, the teacher answers 'yes' and the learner has learnt the language, and if not, the teacher provides a counter-example which is a string that is in the symmetric difference of $L(C)$ and U.

We adapt the L^* algorithm to learn *some* language from a *range* of languages, namely to learn a language that is a superset of L_2 and a subset of L_1. We do not, of course, construct L_1 or L_2 explicitly, but instead answer queries using model-checking queries performed on M_1 and M_2 respectively.

Given an equivalence query with conjecture $L(C)$, the test for equivalence can be split into two— checking the *subset* query $L(C) \subseteq U$ and checking the *superset* query $L(C) \supseteq U$. To check the subset query, we check if $L(C) \subseteq L_1$, and to check the superset query we check whether $L(C) \supseteq L_2$. If these two tests pass, then we declare that the learner has indeed learnt the language as the conjecture is an adequate assumption.

The membership query is more ambiguous to handle. When the learner asks whether a word w is in U, if w is not in L_1, then we can clearly answer in the negative, and if w is in L_2 then we can answer in the affirmative. However, if w is in L_1 but not in L_2, then answering either positively or negatively can rule out certain candidates for A.

In this paper, the strategy we have chosen is to always answer membership queries with respect to L_1. It is possible to explore alternative strategies that involve L_2 also.

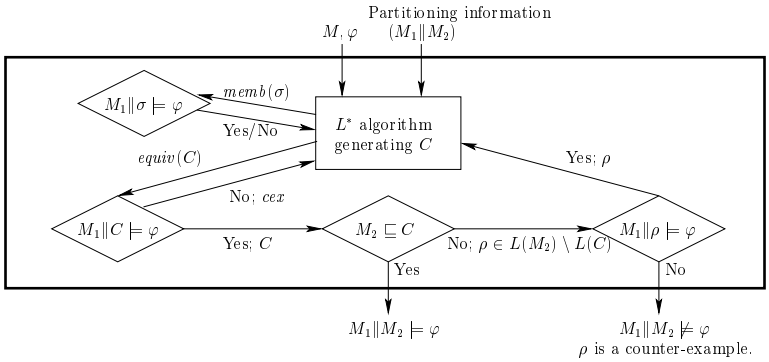

Fig. 1. Overview of compositional verification by learning assumptions

Figure 1 illustrates the high-level overview of our compositional verification procedure. Membership queries are answered by checking safety with respect to M_1. To answer the equivalence query, we first check the subset query (by a safety check with respect to M_1); if the query fails, we return the counterexample found to L^*. If the subset query passes, then we check for the superset query by checking refinement with respect to M_2. If this superset query also passes, then we declare M satisfies φ since C satisfies both premises of the proof rule. Otherwise, we check if the counter-example trace ρ (which is a behavior of M_2 but not in $L(C)$) keeps M_1 safe. If it does not, we conclude that $M_1 \| M_2$ does not satisfy φ; otherwise, we give ρ back to the L^* algorithm as a counter-example to the superset query.

One of the nice properties of the L^* algorithm is that it takes time polynomial in the size of the minimal automaton accepting the learnt language (and polynomial in the lengths of the counter-examples provided by the teacher). Let us now estimate bounds on the size of the automaton constructed by our algorithm, 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 L_1 (membership and subset queries are resolved using L_1 and counter-examples to superset queries, though derived using M_2, are checked for consistency with L_1 before it is passed to the learner).

Now, if $M_1 \| M_2$ does indeed satisfy φ, then L_2 is a subset of L_1 and hence B_1 is an adequate assumption that witnesses the fact that $M_1 \| M_2$ satisfies φ. If $M_1 \| M_2$ does not satisfy φ, then L_2 is not a subset of L_1. Again B_1 is an adequate automaton which if learnt will show that $M_1 \| M_2$ does not satisfy φ (since this assumption when checked with M_2, will result in a run ρ which is exhibited by M_2 but not in L_1, and hence not safe with respect to M_1).

Hence B_1 is an adequate automaton to learn in both cases to answer the model-checking question, and all answers to queries are consistent with B_1. The L^* algorithm has the property that the automata it constructs monotonically grow with each iteration in terms of the number of states, and are always min-

```
1:   R := {ε};  E := {ε};
2:   foreach (a ∈ Σ) { G[ε, ε] := member(ε·ε);  G[ε·a, ε] := member(ε·a·ε);  }
3:   repeat:
4:       while ((r_new := closed(R, E, G)) ≠ null) {
5:           add(R, r_new);
6:           foreach (a ∈ Σ), (e ∈ E) { G[r_new·a, e] := member(r_new·a·e);  }
7:       }
8:       C := makeConjectureMachine(R, E, G);
9:       if ((cex := equivalent(C)) = null) then return C;
10:      else {
11:          e_new := findSuffix(cex);
12:          add(E, e_new);
13:          foreach (r ∈ R), (a ∈ Σ) {
14:              G[r, e_new] := member(r·e_new);  G[r·a, e_new] := member(r·a·e_new);
15:      } }
```

Fig. 2. L^* algorithm

imal. Consequently, we are assured that our procedure will not construct any automaton larger than B_1.

Hence our procedure always halts and reports correctly whether $M_1 \| M_2$ satisfies φ, and in doing so, it never generates any assumption with more states than the minimal DFA accepting L_1.

4 Symbolic implementation of L^* algorithm

4.1 L^* algorithm

The L^* algorithm learns an unknown regular language and generates a minimal DFA that accepts the regular language. This algorithm was introduced by Angluin [6], but we use an improved version by Rivest and Schapire [21]. The algorithm infers the structure of the DFA by asking a teacher, who knows the unknown language, membership and equivalence queries.

Figure 2 illustrates the improved version of L^* algorithm [21]. 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 machine, 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 R and E are sets of strings over Σ, and G is a function from $(R \cup R{\cdot}\Sigma){\cdot}E$ to $\{0, 1\}$. More precisely, R is a set of representative strings for states in the 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 (for any two states of the automaton being built, there is a string in E which is accepted from one and rejected from the other). G maps strings σ in $(R \cup R{\cdot}\Sigma){\cdot}E$ 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

$(R \cup R{\cdot}\Sigma) \cdot E$ (line 2). In line 4, it 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, each such $r \cdot a$ (e.g., r_{new} is $r \cdot a$ in line 5) is simply added to R. The algorithm again updates G with regard to $r{\cdot}a$ (line 6). Once the table is closed, it constructs a conjecture DFA $C = (Q, q_0, F, \delta)$ as follows (line 8): $Q = R$, $q_0 = \varepsilon$, $F = \{r \in R \mid G[r, \varepsilon] = 1\}$, 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 14). Intuitively, the current conjecture machine has guessed wrong since this point. 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}.

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

As we discussed in Section 3, we use the L^* algorithm to identify $A(X_A, X_A^I, X_A^O, Init_A, T_A)$ satisfying the premises of the proof rule, where $X_A^{IO} = X^{IO}$. A is hence a language over the alphabet S^{IO}, and the L^* algorithm can learn A in time polynomial in the size of A (and the counter-examples). However, when we apply the L^* algorithm to analyze a large module (especially when the number of input and output variables is large), the large alphabet size poses many problems: (1) the constructed DFA has too many edges when represented explicitly, (2) the size of the observation table, which is polynomial in Σ and the size of the conjectured automaton, gets very large, and (3) the number of membership queries needed to fill each entry in the observation table also increases. To resolve these problems, we present a symbolic implementation of the L^* algorithm.

4.2 Symbolic implementation

For describing our symbolic implementation for the L^* algorithm, we first explain the essential data structures the algorithm needs, and then present our implicit data structures corresponding to them. The L^* algorithm uses the following data structures:

- `string R[int]`: each `R[i]` is a representative string for i-th state q_i in the conjecture DFA.
- `string E[int]`: each `E[i]` is i-th experiment string.
- `boolean G1[int][int]`: each `G1[i][j]` is the result of the membership query for `R[i]`·`E[j]`.

– `boolean G2[int][int][int]`: 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 of the observation table is split into two arrays, `G1` and `G2`, where `G1` is an array for a function from $R \cdot E$ to $\{0,1\}$ and `G2` is for a function from $R \cdot \Sigma \cdot E$ to $\{0,1\}$. The L^* algorithm initializes the data structures as following: `R[0]=E[0]=`ε, `G1[0][0]=`$member(\varepsilon \cdot \varepsilon)$, and `G2[0][i][0]=`$member(\varepsilon \cdot a_i \cdot \varepsilon)$ (for every $a_i \in \Sigma$). Once it introduces a new state or a new experiment, it adds to `R[]` or `E[]` and updates `G1` and `G2` by membership queries. These arrays also encode 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 implementation, we do not wish to construct `G2` in order to construct conjecture DFAs by explicit membership queries since $|\Sigma|$ is too large. While the explicit L^* algorithm asks for each state r, alphabet symbol a and experiment e, if $r \cdot a \cdot e$ is a member, we compute, given a state r and a *boolean vector* v, the set of alphabet symbols a such that for every $j \leq |v|$, $member(r \cdot a \cdot e_j) = v[j]$. For this, we have the following data structures:

– `int nQ`: the number of states in the current DFA.
– `int nE`: the number of experiment strings.
– `BDD R[int]`: each `R[i]` ($0 \leq$ `i` $<$ `nQ`) is a BDD over X_1 to represent the set of states of the module M_1 that are reachable from an initial state of M_1 by the representative string r_i of the i-th state q_i: $postImage(Init_1(X_1), r_i)$.
– `BDD E[int]`: each `E[i]` ($0 \leq$ `i` $<$ `nE`) is a BDD over X_1 to capture a set of states of M_1 from which some state violating φ is reachable by the i-th experiment string e_i: $preImage(\neg\varphi(X_1), e_i)$.
– `booleanVector G1[int]`: 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`) represents whether $r_i \cdot e_j$ is a member where r_i is a representative string for `R[i]` and e_j is an experiment string for `E[j]`: whether `R[i]` and `E[j]` have empty intersection.
– `booleanVector Cd[int]`: 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 ones are constructed as states of the next conjecture DFA. The `Cd[]` vector describes all these state candidates and each element is the boolean vector of each candidate. |`Cd`| $= 2 \cdot$`nQ`.

Given $M = M_1 \| M_2$ and φ, we initialize the data structures as follows. `R[0]` is the BDD for $Init_1(X_1)$ 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. In addition, we have the following functions that manipulate the above data structures for implementing the L^* algorithm implicitly (Figure 3 illustrates the pseudo-code for the important ones.):

```
BDD edges(int i, booleanVector v){
  BDD eds := true;  // eds is a BDD over X^IO.
  foreach (0 ≤ j < nE){  // In the below, X_1^L = X_1 \ X^IO.
    if (v[j]) then eds := eds ∧ ¬(∃X_1^L, X_1'. R[i](X_1)∧T_1(X_1, X_1^I, X_1')∧E[j](X_1'));
    else eds := eds ∧ (∃X_1^L, X_1'. R[i](X_1) ∧ T_1(X_1, X_1^I, X_1') ∧ E[j](X_1'));
  }
  return eds;
}
void addR(int i, BDD b, booleanVector v){
  BDD io := pickOneState(b);  // io is a BDD representing one alphabet symbol.
  R[nQ] := (∃X_1, X_1^I. R[i](X_1) ∧ io ∧ T_1(X_1, X_1^I, X_1'))[X_1' → X_1];
  G1[nQ++] := v;
}
void addE(BDD[] bs){
  BDD b := φ;  // b is a BDD over X_1.
  for (j := length(bs); j > 0; j--) { b := ∃X_1^I, X_1'. b(X_1') ∧ bs[j] ∧ T_1(X_1, X_1^I, X_1'); }
  E[nE] := ¬b;
  foreach (0 ≤ i < nQ) {
    if ((R[i] ∧ E[nE]) = false) then G1[i][nE] := 1;
    else G1[i][nE] := 0;
    foreach (0 ≤ j < nE) { Cd[2i][j] := G1[i][j];  Cd[2i+1][j] := G1[i][j]; }
    Cd[2i][nE] := 0;   Cd[2i+1][nE] := 1;
  }
  nE++;
}
```

Fig. 3. Symbolic implementation of observation table

- `BDD edges(int, booleanVector)`: this function, given an integer `i` and a boolean vector `v` ($0 \leq$ `i` $<$ `nQ`, $|$`v`$|$ = `nE`), returns a BDD over X^{IO} representing the set of alphabet symbols by which there is an edge from state q_i to a state that has `v` as its boolean vector.
- `void addR(int, BDD, booleanVector)`: when we introduce a new state (whose predecessor state is q_i, the BDD representing edges from q_i is `b` and the boolean vector is `v`), `addR(i, b, v)` updates `R`, `G1` and `nQ`.
- `void addE(BDD[])`: given 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`. It also constructs a new set `Cd[]` of state candidates for the next iteration.
- `boolean isInR(booleanVector)`: given a boolean vector `v`, `isInR(v)` checks whether `v` = `G1[i]` for some `i`.
- `BDD[] findSuffix(BDD[])`: given a counter-example `cex` (from equivalence queries) represented by a BDD array, `findSuffix(cex)` returns a BDD array representing the longest suffix that witnesses the difference between the conjecture DFA and A.

While the L^* algorithm constructs a conjecture machine by computing `G2` and comparing between `G1` and `G2`, we directly make a symbolic conjecture DFA $C(X_C, X^{IO}, Init_C, F_C, T_C)$ with the following components:

- X_C is a set of boolean variables representing states in C ($|X_C| = \lceil \log_2 \texttt{nQ} \rceil$). Valuations of the variables can be encoded from its index for `R`.
- X^{IO} is a set of boolean variables defining its alphabet, which comes from M_1 and M_2.
- $Init_C(X_C)$ is an initial state predicate over X_C. $Init_C(X_C)$ is encoded from the index of the state q_0: $Init_C(X_C) = \bigwedge_{x \in X_C} (x \equiv 0)$.
- $F_C(X_C)$ is a predicate for accepting states. It is encoded from the indices of the states q_i such that `G1[`i`][0]`=1.
- $T_C(X_C, X^{IO}, X'_C)$ is a transition predicate over $X_C \cup X^{IO} \cup X'_C$; that is, if $T_C(i, a, j) = \mathit{true}$, then the DFA has an edge from state q_i to q_j labeled by a. To get this predicate, we compute a set of edges from every state q_i to every state candidate with boolean vector v by calling `edges(i, v)`.

This symbolic DFA $C(X_C, X^{IO}, Init_C, F_C, T_C)$ can be easily converted to a symbolic module $M_C(X_C, X^I, X^O, Init_C, T_C)$. Now, we can construct a symbolic conjecture DFA C using implicit membership queries by `edges()`. In addition, we have the following functions for equivalence queries:

- `BDD[] subsetQ(SymbolicDFA)`: our subset query is to check whether all strings *allowed by* C make M_1 stay in states satisfying φ. Hence, given a symbolic DFA $C(X_C, X^{IO}, Init_C, F_C, T_C)$, we check $M_1 \| M_C \models (F_C \rightarrow \varphi)$ by reachability checking, where M_C is a symbolic module converted from C. If so, it returns *null*; otherwise, it returns a BDD array as a counter-example.
- `BDD[] supersetQ(SymbolicDFA)`: it checks that $M_2 \sqsubseteq C$. The return value is similar with `subsetQ()`. Since C is again a (symbolic) DFA, we can simply implement it by symbolic reachability computation for the product of M_2 and M_C. If it reaches the non-accepting state of C, the sequence reaching the non-accepting state is a witness showing $M_2 \not\sqsubseteq C$.
- `boolean safeM1(BDD [])`: given a string σ represented by a BDD array, it executes M_1 according to σ. If the execution reaches a state violating φ, it returns *false*; otherwise, returns *true*.

Figure 4 illustrates our symbolic compositional verification (SCV) algorithm. We initialize `nQ, nE, R, E, G1, Cd` and C in lines 1–3. We then compute a set of edges (a BDD) from every source state q_i to every state candidate with boolean vector `Cd[j]`. Once we reach a new state, we update `R, nQ` and `G1` by `addR()` (line 9). This step makes the conjecture machine closed. If we have a non-empty edge set by `edges()`, then we update the conjecture C (line 10). After constructing a conjecture DFA, we ask an equivalence query as discussed in Section 3 (lines 12–15). If we cannot conclude true nor false from the query, we are provided a counter-example from the teacher and get a new experiment string from the counter-example. `E, nE, Cd` and `G1` are then updated based on the new experiment string. We implement this algorithm with the BDD package in a symbolic model checker NuSMV.

```
boolean SCV(M1, M2, φ)
1:    nQ := 1;  nE := 1;  R[0] := Init1(X1);  E[0] := ¬φ;
2:    G1[0][0] := 1;  Cd[0] := 0;  Cd[1] := 1;
3:    C := initializeC();
4:    repeat:
5:        foreach (0 ≤ i < nQ) {
6:            foreach (0 ≤ j < 2·nQ) {
7:                eds := edges(i, Cd[j]);
8:                if (eds ≠ false) then {
9:                    if (¬isInR(Cd[j])) then addR(i, eds, Cd[j]);
10:                   C := updateC(i, eds, indexofR(Cd[j]));
11:       } } }
12:       if ((cex := subsetQ(C)) = null) then {
13:           if ((cex := supersetQ(C) = null) then return true;
14:           else if (¬safeM1(cex)) then return false;
15:       }
16:       addE(findSuffix(cex));
```

Fig. 4. Symbolic compositional verification algorithm

5 Experiments

We first explain an artificial example (called '`simple`') to illustrate our method and then report results on '`simple`' and four examples from the NuSMV package.

Example: simple Module M_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 dummy array that does not affect x at all. Module M_2 has a variable y (initially set to 0 and is never updated) and also a dummy array that does not affect y at all. For $M_1 \| M_2$, we want to check that x is always 0. Both dummy arrays are from an example *swap* known to be hard for BDD encoding [18]. Our tool explores M_1 and M_2 separately with a two-state assumption (which allows only $y = 0$), while ordinary model checkers will search whole state space of $M_1 \| M_2$.

For some examples from the NuSMV package, we slightly modified them because our tool does not support the full syntax of the NuSMV language. The primary selection criterion was to include examples for which NuSMV takes a long time or fails to complete. All experiments were performed on a Sun-Blade-1000 workstation using 1GB memory and SunOS 5.9. The results for the examples are shown in Table 1. We compare our symbolic compositional verification tool (SCV) with the invariant checking (with early termination) of NuSMV 2.2.2. The table has the number of variables in total, in M_1, in M_2 and the number of input/output variables between the modules, execution time in seconds, the peak BDD size and the number of states in the assumption we learn (for SCV). Entries denoted '–' mean that a tool did not complete within 2 hours.

The results of `simple` are also shown in Table 1. For `simple1` through `simple4`, we just increased the size of dummy arrays from 8 to 11, and checked

example name	spec	tot var	M_1 var	M_2 var	IO var	SCV			NuSMV	
						time	peak BDD	assumption states	time	peak BDD
simple1		69	36	33	4	19.2	607,068	2	269	3,993,976
simple2	true	78	41	37	5	106	828,842	2	4032	32,934,972
simple3		86	45	41	5	754	3,668,980	2	–	–
simple4		94	49	45	5	4601	12,450,004	2	–	–
guidance1	false	135	24	111	23	124	686,784	20	–	–
guidance2	true	122	24	98	22	196	1,052,660	2	–	–
guidance3	true	122	58	64	46	357	619,332	2	–	–
barrel1	false					20.3	345,436	3	1201	28,118,286
barrel2	true	60	30	30	10	23.4	472,164	4	4886	36,521,170
barrel3	true					–	–	too many	–	–
msi1		45	26	19	25	2.1	289,226	2	157	1,554,462
msi2	true	57	26	31	25	37.0	619,332	2	3324	16,183,370
msi3		70	26	44	26	1183	6,991,502	2	–	–
robot1	false	92	8	84	12	1271	4,169,760	11	654	2,729,762
robot2	true	92	22	70	12	1604	2,804,368	42	1039	1,117,046

Table 1. Experimental results

the same specification. As we expected, SCV generated a 2-state assumption and performed significantly better than NuSMV.

The second example, `guidance`, is a model of a space shuttle digital autopilot. We added redundant variables to M_1 and M_2 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: `guidance1, guidance2, guidance3` have the same models but have different specifications. For `guidance1`, our tool found a counter-example with an assumption having 20 states (If this assumption had been explicitly constructed, the 23 I/O variables would have caused way too many edges to store explicitly).

The third set, `barrel`, is an example for bounded model checking and no variable ordering works well for BDD-based tools. `barrel1` has an invariant derived from the original, but `barrel2` and `barrel3` have our own ones. `barrel1, barrel2` and `barrel3` 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: `msi1` has 3 nodes, `msi2` has 4 nodes and `msi3` has 5 nodes. They have the same specification that is related to only two nodes, and we fixed the same component M_1 in all of them. As the number of nodes grew, NuSMV required much more time and the BDD sizes grew more quickly than in our tool.

`robot1` and `robot2` are robotics controller models and we again added redundant variables to M_1 and M_2, as in the case of `guidance` example. Even though SCV took more time, this example shows that SCV can be applied to models for which non-trivial assumptions are needed. More details about the examples are available at `http://www.cis.upenn.edu/~wnam/cav05/`.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM TOPLAS*, 17:507–534, 1995.
2. R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proc. 32nd ACM POPL*, pages 98–109, 2005.
3. R. Alur, L. de Alfaro, T.A. Henzinger, and F. Mang. Automating modular verification. In *CONCUR'99: Concurrency Theory*, LNCS 1664, pages 82–97, 1999.
4. R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. A preliminary version appears in *Proc. 11th LICS, 1996.*
5. R. Alur, T.A. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *10th CAV*, pages 516–520, 1998.
6. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.
7. H. Barringer, C.S. Pasareanu, and D. Giannakopolou. Proof rules for automated compositional verification through learning. In *Proc. 2nd SVCBS*, 2003.
8. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. 5th TACAS*, pages 193–207, 1999.
9. R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
10. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. CAV 2002*, LNCS 2404, pages 359–364, 2002.
11. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
12. J.M. Cobleigh, D. Giannakopoulou, and C.S. Pasareanu. Learning assumptions for compositional verification. In *Proc. 9th TACAS*, LNCS 2619, pages 331–346, 2003.
13. O. Grümberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
14. T.A. Henzinger, S. Qadeer, and S. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proc. CAV 98*, LNCS 1427, pages 521–525, 1998.
15. R.P. Kurshan. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press, 1994.
16. K.L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.
17. K.L. McMillan. A compositional rule for hardware design refinement. In *CAV 97: Computer-Aided Verification*, LNCS 1254, pages 24–35, 1997.
18. K.L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. 14th Computer Aided Verification*, LNCS 2404, pages 250–264, 2002.
19. K.S. Namjoshi and R.J. Trefler. On the completeness of compositional reasoning. In *CAV 2000: Computer-Aided Verification*, LNCS 1855, pages 139–153, 2000.
20. D. Peled, M.Y. Vardi and M. Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2): 225-246, 2002.
21. R.L. Rivest and R.E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.
22. E.W. Stark. A proof technique for rely-guarantee properties. In *FST & TCS 85*, LNCS 206, pages 369–391, 1985.
23. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety properties for FIFO automata. In *Proc. 24th FSTTCS*, pages 494–505, 2004.