PAPER

Special Section on Formal Approaches

On Locally Minimum and Strongest Assumption Generation Method for Component-Based Software Verification

Hoang-Viet TRAN(a), Nonmember, Ngoc Hung PHAM(b), Member, and Viet Ha NGUYEN(c), Nonmember

SUMMARY Since software becomes more complex during its life cycle, the verification cost becomes higher, especially for such methods which are using model checking in general and assume-guarantee reasoning in specific. To address the problem of reducing the assume-guarantee verification cost, this paper presents a method to generate locally minimum and strongest assumptions for verification of component-based software. For this purpose, we integrate a variant of membership queries answering technique to an algorithm which considers candidate assumptions that are smaller and stronger first, larger and weaker later. Because the algorithm stops as soon as it reaches a conclusive result, the generated assumptions are the locally minimum and strongest ones. The correctness proof of the proposed algorithm is also included in the paper. An implemented tool, test data, and experimental results are presented and discussed.

key words: assume-guarantee reasoning, model checking, software verification, locally minimum assumption, locally strongest assumption

1. Introduction

As high demand on quality in modern software industry has increased while software becomes more and more complex during its life cycle, the cost for software verification has increased dramatically. As a result, reducing the cost of software verification becomes a must for most of software companies, especially who are using the model checking methods[5], [7], [26]. In addition, a well known problem of model checking is the state space explosion[5], [26] which makes model checking a hard approach to be applied in practice. This leads to the introduction of the assume-guarantee reasoning verification method[6], [8], [10], [11], [13], [19], [24], which is believed to be a promising solution for dealing with the state space explosion problem of model checking thanks to its divide-and-conquer strategy. However, the cost of assume-guarantee verification remains high that prevents the method from being applied effectively for large-scale software in practice. Consequently, reducing assume-guarantee verification cost plays an important role in software verification in specific and in software development in general.

Recently, there are many works that are relative to improvement of the assume-guarantee reasoning method for component-based software (CBS)[8], [10]. These works include some variant of assume-guarantee rules[2], [22], symbolic implementation for assume-guarantee rules[22], several improvements[3], [4], [12], [28], an extension to support liveness properties[9], etc. However, considering the assume-guarantee rules $M_1 \parallel A \models p$ and $M_2 \models A$, where $M_1$, $M_2$ and $p$ are software components and the safety property under checking, we can see that these rules are actually the language emptiness problem of $M_1 \parallel A \models p_{err}$ and $M_2 \models A_{err}$, where $p_{err}$ and $A_{err}$ are labeled transition systems which trap possible violations of $p$ and $A$ with the error state, respectively[29]. Therefore, the smaller size the assumption $A$ is, the more verification cost we can reduce for software verification process. Moreover, with two assumptions that have the same size, the stronger assumption will reduce more verification cost than the weaker one thanks to the fact that satisfiability is actually the language containment problem, i.e., $M \models p \equiv L(M) \subseteq L(p)$. However, the $L$-based assumption generation method proposed by Giannakopoulou et al.[10] and Cobleigh et al.[8] stops as soon as it reaches the conclusive result, the algorithm is not able to find either the smallest assumptions or strongest ones.

Hung et al. published a series of researches that proposed a potential method to reduce CBS verification cost by generating minimum assumptions[15], [17], [18] and use those generated minimum assumptions in the context of software evolution[14], [16]. However, the proposed method based on search algorithms such as breadth-first search[15] and depth-limited search[17], [18] in the space of closed observation tables to find the minimum satisfied assumptions. In addition, those methods proposed by Hung et al. have not taken the languages of the generated assumptions in consideration. As a result, such methods have not optimally reduced the verification cost for CBS.

This paper presents a different method from the method proposed by Hung et al. that can generate locally minimum and strongest assumptions. We check every possibility that a trace $s$ does not belong to the language of the assumption to be generated. For this purpose, we use a variant of the membership queries answering technique and integrate it to an algorithm to generate assumptions. Further more, we use candidate assumptions generated by $L$-based assumption generation method proposed by Cobleigh et al.[8] as baseline candidate assumptions for analysis. In order to have the minimum assumptions, we try every possibility that the assumption being generated has one state to the number of states of the baseline candidate assumption $A_r$. That is to get
t-combination of states from the set of states of $A_t$, where $t$ is from 1 to $|A_t|$. Then, among the candidate assumptions which have $t$ states, let consider a candidate assumption $C$ and assume that we have $n$ "?” membership queries results in the observation table that is corresponding to $C$. We try every possibility that $k$ “?” results are false (the corresponding traces do not belong to $C$), where $k$ is from $n - 1$ to 1. This means that we try from 1 trace to $n - 1$ traces belong to $L(C)$. By using this method to check candidate assumptions, we will get the minimum assumption first, and large ones later. In addition, for a candidate assumption with $t$ states, because we tried the possibility that $n - 1$ traces which are corresponding to “?” results do not belong to $L(C)$ first, less number of traces later, we will get the strongest assumption first, and the weaker assumption later. As a result, the generated assumption by our method will be the locally minimum and strongest one.

The rest of this paper is organized as follows. We review some background concepts in Sect. 2. Section 3 presents the $L^*$-based assumption generation method. Later, the algorithm which generates locally minimum and strongest assumption will be shown in Sect. 4. The correctness of the proposed algorithm is also discussed in Sect. 4. An implemented tool and experimental results are shown and discussed in Sect. 5. In the end, we conclude the paper in Sect. 7.

2. Background

In this section, we present some basic concepts which will be used in this work.

LTSs. This research uses Labeled Transition Systems (LTSs) to model behaviors of components. Let $\mathcal{Act}$ be the universal set of observable actions and let $\tau$ denote a local action unobservable to a component environment. We use $\pi$ to denote a special error state. An LTS is defined as follows.

Definition 1. (LTS). An LTS $M$ is a quadruple $\langle Q, \Sigma, \delta, q_0 \rangle$, where:

- $Q$ is a non-empty set of states,
- $\Sigma \subseteq \mathcal{Act}$ is a finite set of observable actions called the alphabet of $M$,
- $\delta \subseteq Q \times \Sigma \times \{\tau\} \times Q$ is a transition relation, and
- $q_0 \in Q$ is the initial state.

The size of an LTS $M = \langle Q, \Sigma, \delta, q_0 \rangle$ is the number of states of $M$, denoted by $|M|$ (i.e., $|M| = |Q|$).

A trace $\sigma$ of an LTS $M = \langle Q, \Sigma, \delta, q_0 \rangle$ is a finite sequence of actions $a_1a_2\ldots a_n$ such that there exists a sequence of states starting at the initial state (i.e., $q_0q_1\ldots q_n$) such that for $1 \leq i \leq n$, $(q_{i-1}, a_i, q_i) \in \delta$, $q_i \in Q$.

Given two sets of event sequences $P$ and $Q$, we define the concatenation operator $P \cdot Q = \{pq \mid p \in P, q \in Q\}$, where $pq$ presents the concatenation of the event sequences $p$ and $q$.

We call the set of all traces of $M$ the language of $M$, denoted by $L(M)$. Let $\sigma = a_1a_2\ldots a_n$ be a finite trace of an LTS $M$. We use $[\sigma]$ to denote the LTS $M_{\sigma} = \langle Q, \Sigma, \delta, q_0 \rangle$ with $Q = \{q_0, q_1, \ldots, q_n\}$, and $\delta = \{(q_{i-1}, a_i, q_i)\}$, where $1 \leq i \leq n$.

Parallel Composition. The parallel composition operator $\parallel$ is a commutative and associative operator up-to language equivalence that combines the behavior of two models by synchronizing the common actions to their alphabets and interleaving the remaining actions.

Definition 2. (Parallel composition operator). The parallel composition between $M_1 = \langle Q_1, \Sigma_M, \delta_1, q_0 \rangle$ and $M_2 = \langle Q_2, \Sigma_M, \delta_2, q_0' \rangle$, denoted by $M_1 \parallel M_2$, is defined as follows. $M_1 \parallel M_2$ is equivalent to $\parallel$ if either $M_1$ or $M_2$ is equivalent to $\parallel$, where $\parallel$ denotes the LTS $\langle |\pi|, \mathcal{Act}, \emptyset, \pi \rangle$. Otherwise, $M_1 \parallel M_2$ is an LTS $M = \langle Q, \Sigma, \delta, q_0 \rangle$ where $Q = Q_1 \times Q_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $q_0 = (q_0', q_0)$, and the transition relation $\delta$ is given by the following rules:

1. $\delta = \Sigma_1 \cup \Sigma_2$.
2. $\delta = \delta_1 \cup \delta_2$.
3. $\delta = \delta_1 \cup \delta_2$.

Safety LTSs, Safety Property, Satisfiability and Error LTSs. A safety LTS is a deterministic LTS that contains no state that is equivalent to $\pi$ state. A safety property asserts that nothing bad happens for all time. A safety property $p$ is specified as a safety LTS $p = \langle Q, \Sigma_p, \delta, q_0 \rangle$ whose language $L(p)$ defines the set of acceptable behaviors over $\Sigma_p$.

Definition 3. (Satisfiability). An LTS $M$ satisfies a required property $p$, denoted by $M \models p$, if and only if $\forall \sigma \in L(M)$: $(\sigma_{\Sigma_p} \models L(p))$, where $\sigma_{\Sigma_p}$ denotes the trace obtained by removing from $\sigma$ all occurrences of actions $a \notin \Sigma_p$.

Remark 1. When we check whether an LTS $M$ satisfies a required property $p$, an error LTS, denoted by $p_{err}$, is created which traps possible violations with the $\pi$ state. $p_{err}$ of a property $p = \langle Q, \Sigma_p, \delta, q_0 \rangle$ is $\langle Q \cup \{\pi\}, \Sigma_p, \delta', q_0 \rangle$, where $\delta' = \delta \cup \{(q, a, \pi) \mid a \in \Sigma_p \}$.

From the above definition, we can see that the error LTSs are complete, meaning each state other than the error state has outgoing transitions for every action in the alphabet. In order to verify a component $M$ satisfying a property $p$, both $M$ and $p$ are represented by safety LTSs, the parallel compositional system $M \parallel p_{err}$, is then computed. If some states $(q, \pi)$ are reachable in the compositional system, $M$ violates $p$. Otherwise, it satisfies $p$.

Definition 4. (Deterministic finite state automata - DFA). A DFA $D$ is a five tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where:

- $Q, \Sigma, \delta, q_0$ are defined as for deterministic LTSs, and
- $F \subseteq Q$ is a set of accepting states.

Let $D$ be a DFA and $\sigma$ be a string over $\Sigma$. We use
\(\delta(q, \sigma)\) to denote the state that \(D\) will be in after reading \(\sigma\) starting from the state \(q\). A string \(\sigma\) is accepted by a DFA \(D = (Q, \Sigma, \delta, q_0, F)\) if \(\delta(q_0, \sigma) \in F\). The set of all string \(\sigma\) accepted by \(D\) is called the language of \(D\) (denoted by \(L(D)\)). Formally \(L(D) = \{\sigma \mid \delta(q_0, \sigma) \in F\}\).

**Definition 5.** (Assume-Guarantee Reasoning). Let \(M\) be a system which consists of two components \(M_1\) and \(M_2\), \(p\) be a safety property, and \(A\) be an assumption about \(M_1\)'s environment. The assume-guarantee rules are described as following formula [8].

\[
\begin{align*}
\text{(Step1)} & \quad (A) M_1 (p) \\
\text{(Step2)} & \quad (true) M_2 (A)
\end{align*}
\]

We use the formula \((true) M (A)\) to represent the compositional formula \(M||A_{err}\). The formula \((A) M (p)\) is true if whenever \(M\) is part of a system satisfying \(A\), then the system must also guarantee \(p\). In order to check the formula, where both \(A\) and \(p\) are safety LTSs, we compute the compositional formula \(A||M||p_{err}\) and check if the error state \(\pi\) is reachable in the composition. If it is, the formula is violated. Otherwise it is satisfied.

**Definition 6.** (Weakest Assumption) [10]. The weakest assumption, denoted by \(A_{W}\), describes exactly those traces over the alphabet \(\Sigma = (\Sigma_M \cup \Sigma_\sigma) \cap \Sigma_M\), which, the error state \(\pi\) is not reachable in the compositional system \(M||p_{err}\). \(A_{W}\) means that for any environment component \(E\), \(M_1||E \models p\) if and only if \(E \models A_{W}\).

**Definition 7.** (Strongest Assumption). Let \(A_S\) be an assumption that satisfies the assume-guarantee rules. If for all \(A\) satisfying the assume-guarantee rules: \(L(A_S) \subseteq L(A)\), we call \(A_S\) the strongest assumption.

**Note 1.** Let \(\mathcal{A}\) be a subset of assumptions that satisfy the assume-guarantee rules and \(A_{LS} \subseteq \mathcal{A}\). If for all \(A \in \mathcal{A}\): \(L(A_{LS}) \subseteq L(A)\), we call \(A_{LS}\) the locally minimum assumption.

**Definition 8.** (Minimum Assumption). Let \(A_M\) be an assumption that satisfies the assume-guarantee rules. If for all \(A\) satisfying the assume-guarantee rules: \(|A_M| \leq |A|\), we call \(A_M\) the minimum assumption.

**Note 2.** Let \(\mathcal{A}\) be a subset of assumptions that satisfy the assume-guarantee rules and \(A_{LAM} \subseteq \mathcal{A}\). If for all \(A \in \mathcal{A}\): \(|A_{LAM}| \leq |A|\), we call \(A_{LAM}\) the locally minimum assumption.

**Definition 9.** (Observation table). Given a set of alphabet symbols \(\Sigma\), an observation table is a 3-tuple \((S, E, T)\), where:

- \(S \subseteq \Sigma\) is a set of prefixes,
- \(E \subseteq \Sigma\) is a set of suffixes, and
- \(T : (S \cup \Sigma, E) \rightarrow \{true, false\}\). With a string \(s \in \Sigma\), \(T(s) = true\) means \(s \in L(A)\), otherwise \(s \notin L(A)\), where \(A\) is the corresponding assumption to \((S, E, T)\).

An observation table is closed if \(\forall s \in S, \forall a \in \Sigma, \exists s' \in S, \forall e \in E: T(sae) = T(s'e)\). In this case, \(s'\) presents the next state from \(s\) after seeing \(a\), \(sa\) is indistinguishable from \(s'\) by any of suffixes. Intuitively, an observation table \((S, E, T)\) is closed means that every row \(sa \in S\) has a matching row \(s'\) in \(S\).

When an observation table \((S, E, T)\) over an alphabet \(\Sigma\) is closed, we define the corresponding DFA that accepts the associated language as follows [1]. \(M = (Q, \Sigma_M, \delta, q_0, F)\), where:

- \(Q = \{row(s) \mid s \in S\}\),
- \(q_0 = row(\lambda)\),
- \(F = \{row(s) \mid s \in S\} \cup \{T(s) = true\}\),
- \(\Sigma_M = \Sigma\), and
- \(\delta(row(s), a) = row(s.a)\).

From this way of constructing DFA from an observation table \((S, E, T)\), we can see that each states of the DFA which is being created is corresponding to one row in \(S\). Therefore, from now on, we sometimes call the rows in \((S, E, T)\) its states.

**Remark 2.** The DFAs generated from observation table in this context are complete, minimal, and prefix-closed (an automaton \(D\) is prefix-closed if \(L(D)\) is prefix-closed, i.e., for every \(\sigma \in L(D)\), every prefix of \(\sigma\) is also in \(L(D)\)). Therefore, these DFAs contain a single non-accepting state (denoted by \(nas\)) [8]. Consider a DFA \(D = (Q \cup \{nas\}, \Sigma, \delta, q_0, Q)\) in this context, we can calculate the corresponding safety LTS \(A_{S}\) by removing the non-accepting state \(nas\) and all of its ingoing transitions. Formally, we have \(A = \langle Q, \Sigma, \delta \cap (Q \times \Sigma \times \{nas\}, q_0)\rangle\).

### 3. The \(L^*\)–Based Assumption Generation Method

#### 3.1 The \(L^*\) Algorithm

\(L^*\) algorithm was developed by Angluin [1] and later improved by Rivest and Schapire [27]. In this paper, we will refer to the improved version by using the name of the original algorithm, \(L^*\). \(L^*\) algorithm can incrementally learn an unknown regular language by generating a deterministic finite automaton (DFA) that accepts it. The key idea of \(L^*\) algorithm bases on “Myhill Nerode Theorem” [23] in the formal language theory. That is for every regular set \(U \subseteq \Sigma^*\), there exists a unique, minimal deterministic automaton whose states are isomorphic to the set of equivalence classes of the following relation: \(w \equiv u^\prime\) if and only if \(\forall u \in \Sigma^*: uu\epsilon U \Leftrightarrow u^\prime u \in U\). As a result, the main point of \(L^*\) is to learn equivalence classes, i.e., two prefixes are not in the same class if and only if there is a distinguishing suffix \(u\).

Let \(U\) be an arbitrary unknown regular language over some alphabet \(\Sigma\). \(L^*\) will produce a DFA \(D\) such that \(L(D) = U\). For this purpose, the learning process, shown in Fig. 1, is performed by the interaction between two oracles the learner (i.e., \(L^*\)) and the teacher. The teacher is the
3.2 Generating Assumption Using \( L^* \) Algorithm

Given a CBS \( M \) that consists of two components \( M_1 \) and \( M_2 \) (\( M = M_1 \parallel M_2 \)) and a safety property \( p \). The purpose of the assume-guarantee verification is to verify if \( M \models p \) without actually composing \( M_1 \) with \( M_2 \). For this purpose, the \( L^* \)-based assumption generation algorithm [8] generates a contextual assumption that satisfies the assume-guarantee rules using \( L^* \) algorithm [1]. If an assumption \( A \) exists, then \( M \models p \). Otherwise, \( M \not\models p \).

The details of this algorithm are shown in Algorithm 1. For learning an assumption \( A \), Algorithm 1 manages an observation table \((S, E, T)\). The algorithm starts by initializing \( S \) and \( E \) with the empty string \( \lambda \) (line 2). The algorithm then updates \((S, E, T)\) by using membership queries (line 4). While the observation table \((S, E, T)\) is not closed, the algorithm continues adding \( s \) to \( S \) and updating the observation table to make it closed (from line 5 to line 8). When the observation table is closed, the algorithm creates a conjecture \( C \) from the closed table \((S, E, T)\) and asks an equivalence query to the teacher (from line 9 to line 10). The algorithm then stores the result of candidate query to \( \text{equiResult} \). An equivalence query result contains two properties: \( \text{Key} \in \{\text{YES, NO, UNSAT}\} \) (i.e., \( \text{YES} \) means that the corresponding candidate assumption satisfies the assume-guarantee rules; \( \text{NO} \) means that the corresponding candidate assumption does not satisfy assume-guarantee rules, however, at this point, we could not decide if the given system \( M \) does not satisfy \( p \) yet, we can use the corresponding counterexample \( \text{cex} \) to generate a new candidate assumption; \( \text{UNSAT} \) means the given system \( M \) does not satisfy \( p \) and the counterexample is \( \text{cex} \)). One who can answer the following two types of queries from the learner.

- Membership queries: Given a string \( \sigma \in \Sigma^* \) (i.e., “\( \sigma \in U? \)”), the teacher must be able to answer the learner with \text{true} \) if \( \sigma \in U \), and \text{false} \) otherwise.
- Equivalence queries: Given a candidate DFA \( D \) whose language is believed to be identical to \( U \) (“\( L(D) = U? \)”). The teacher must be able to answer the learner with \text{YES} \) if \( L(D) = U \). Otherwise, the teacher must answer the learner with \text{NO} \) and a counterexample \( \text{cex} \) which is a string in the symmetric difference of \( L(D) \) and \( U \).

### Algorithm 1: \( L^* \)-based assumption generation.

```plaintext
begin
  Let \( S = E = \{\lambda\} \)
  while true do
    Update \( T \) using membership queries
    while \((S, E, T)\) is not closed do
      Add \( s \) to \( S \) to make \((S, E, T)\) closed where \( s \in S \) and \( a \in \Sigma \)
      Update \( T \) using membership queries
    end
    Create a conjecture \( C \) from \((S, E, T)\)
    if \( \text{equiResult}.\text{Key} \) is \text{YES} then
      return \( C \)
    else if \( \text{equiResult}.\text{Key} \) is \text{UNSAT} then
      return \text{UNSAT} + \text{cex}
    else
      /* Teacher returns \text{NO} + \text{cex} */
      Add \( e \in \Sigma \) that witnesses the counterexample to \( E \)
    end
  end
end
```

### Fig. 2 Incremental compositional verification during iteration \( i^{th} \).

\( \text{YES} \) \( \text{(i.e.,} \ C \text{is the needed assumption)} \), the algorithm stops and returns \( C \) (line 12). If \( \text{equiResult}.\text{Key} \) is \text{UNSAT} \), the algorithm will stops and returns \text{UNSAT} and \text{cex} \) is the corresponding counterexample. Otherwise, if \( \text{equiResult}.\text{Key} \) is \text{NO} \), it analyzes the returned counterexample \( \text{cex} \) to find a suitable suffixes \( e \). This suffix \( e \) must be such that adding it to \( E \) will cause the next assumption candidate to reflect the difference and keep the set of suffix \( E \) closed. The method to find \( e \) is not in the scope of this paper, please find more details in the improved version of \( L^* \) proposed by Rivest et al. [27]. It then adds \( e \) to \( E \) (line 16) and continues the learning process again from line 4. The incremental composition verification during the iteration \( i^{th} \) is shown in Fig. 2.

In order to answer a membership query whether a trace \( \sigma = a_1 a_2 \ldots a_n \) belongs to \( L(A) \), we create an LTS \( [\sigma] = \langle Q, \Sigma, \delta, q_0 \rangle \) with \( Q = \{q_0, q_1, \ldots, q_n\} \), and \( \delta = \{(q_{i-1}, a_i, q_i)\} \), where \( 1 \leq i \leq n \). The teacher then checks the formula \( \langle[\sigma]\rangle M_1(p) \) by computing compositional system \( [\sigma]|M_1|p_{\text{ref}} \). If the error state \( \pi \) is unreachable, the teacher returns \text{true} \( \text{(i.e.,} \sigma \notin L(A)) \). Otherwise, the teacher returns \text{false} \( \text{(i.e.,} \sigma \in L(A)) \).

In regards to solving equivalence queries, as mentioned in Sect. 3.1, these queries are handled in the teacher by comparing \( L(A) = U \). However, in case of assume-guarantee
reasoning, we have not known what $U$ is yet. The only thing we know is that the assumption $A$ to be generated must satisfy the assume-guarantee rules. Therefore, instead of checking $L(A) = U$, we check if $A$ satisfies the assume-guarantee rules. For more detailed information about when the teacher returns $YES$, $NO$, and $UNSAT$, please refer to $L^\ast$-based assumption generation method proposed by Cobleigh et al. [8].

4. Locally Minimum Assumption Generation Method

This section presents an algorithm that can generate the locally minimum and strongest assumptions for verification of component-based software. The key idea of this algorithm comes from an observation that for a membership query of an arbitrary string $s$, the teacher can check and answer the learner with $true$ or $false$. The original idea behind the teacher’s answer is to check $s$ against the language of the weakest assumption $A_W$. That is, if $s \in L(A_W)$, the answer is $true$. Otherwise, the answer is false [8]. When the teacher answers $true$ to the learner, the teacher has not known whether $s$ actually belongs to the language of the assumption to be generated. The relationship between $s$, the language of the assumption to be generated $L(A)$ and the language of the weakest assumption $L(A_W)$ is shown in Fig. 3.

The original idea of the paper is to generate the minimum and strongest assumptions (i.e., assumptions with the smallest number of states). However, in the space of assumptions that satisfy the assume-guarantee rules, there can be a lot of assumptions. Although we can check whether a string belongs to the language of the weakest assumption, to our knowledge, we have not had the method to find out all satisfied assumptions. That is the reason why we can only have the locally minimum and strongest assumption for a class of assumptions generated by an arbitrary given assumption generation algorithm.

For the purpose of generating the locally minimum and strongest assumption for verification of CBS, we first reuse the variant to the membership queries answering technique from Hung et al. series of papers [15], [17], [18]. Later, we integrate this technique into the proposed algorithm to generate the assumptions with smaller size and language than the ones generated by the algorithm presented in Sect. 3. Then, we prove that the assumption generated by our proposed algorithm is the locally minimum and strongest one. The correctness proof of the proposed algorithm is given in Sect. 4.4.

4.1 A Variant of The Membership Queries Answering Technique

The membership queries answering technique used in $L^\ast$-based assumption generation method in Sect. 3 bases on the language of the weakest assumption $L(A_W)$. Thus, for an arbitrary string $s$, if $\langle s \rangle M_I(p)$, the technique returns $true$ no matter if $s$ actually belongs to the language of the assumption to be generated or not. Because of this observation, the variant of membership queries answering technique will not return true when $s \in L(A_W)$. It will return “?” instead. When the learner receives “?” result, it will consider “?” as false (i.e., the corresponding string does not belong to $L(A)$) and try to learn $A$. If the learner cannot find the satisfied assumption, it will come back one step before, consider the corresponding “?” as true and try to learn assumption again. Details about the variant technique is shown in Algorithm 2. In this variant algorithm when the teacher receives a membership query for a trace $s = a_0a_1...a_n \in \Sigma^*$, it first builds an LTS $[s]$. It then model checks $\langle s \rangle M_I(p)$. If true is returned (i.e., $s \in L(A_W)$), the teacher returns “?” (line 3), and false, otherwise (line 5).

4.2 The Locally Minimum and Strongest Assumption Generation Method

This section presents an algorithm that generates the locally minimum and strongest assumptions which is based on the variant membership queries answering technique presented in Sect. 4.1. The key idea of the algorithm is a series of “try” steps where candidate assumptions generated by $L^\ast$-based assumption generation method are used as baselines for the learning process (denoted by the baseline assumption). At each step of the $L^\ast$-based assumption generation method’s learning process where the observation table is closed, i.e., a candidate assumption can be generated and submitted to an equivalence query, the algorithm will try to see if there can be any other candidate assumption that has less than or equal number of states to that of the baseline candidate assumption with the consideration that “?” can be false. If there is, that assumption will be considered for asking an equivalence query before the baseline candidate assumption. Details of the algorithm are shown in Algorithm 3. The algorithm initializes the observation table $(S, E, T)$

![Fig. 3](image_url) The relationship between $s$, $L(A)$, and $L(A_W)$. 

<table>
<thead>
<tr>
<th>Algorithm 2: Answering membership queries.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong>: A trace $s = a_0a_1...a_n$</td>
</tr>
<tr>
<td><strong>output</strong>: If $s \in L(A_W)$ then “?”, otherwise false</td>
</tr>
<tr>
<td><strong>begin</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>
Algorithm 3: Learning locally minimum & strongest assumptions.

```
begin
  Let $S = E = \{\lambda\}$
  while true do
    Update $T$ using membership queries
    while $(S, E, T)$ is not closed do
      Add $a\tau$ to $S$ to make $(S, E, T)$ closed
      where $s \in S$ and $a \in \Sigma$
      Update $T$ using membership queries
    end
    Consider all “?” results in $(S, E, T)$ as true
    Make the conjecture $A_i$ from $(S, E, T)$
    $tempQ \leftarrow A_i, Q \setminus \{A_i, q_0\}; m \leftarrow |tempQ|
    for each $t$ from 0 to $m$ do
      for each comFinalState = $t$–combination of $m$ states of tempQ do
        Update “?” results to true in rows in $(S, E, T)$ which are not corresponding to any state in comFinalState
        $n \leftarrow 1$ [“?”]
        for each $k$ from 0 to 1 do
          Get one $k$–combination of $n$ “?” results.
          Turn all those “?” results to false, other “?” results are turned to true.
          if The corresponding observation table $(S, E, T)$ is closed then
            Create a candidate assumption $C_{ikj}$. result $\leftarrow$ Ask an equivalence query for $C_{ikj}$.
            if result.Key is YES then
              result.Assumption
            end
            end
          end
        end
      end
    end
    equiResult $\leftarrow$ ask an equivalence query for $A_i$
    if equiResult.Key is YES then
      return $A_i$
    else if equiResult.Key is UNSAT then
      return UNSAT + cex
    else
      /* Teacher returns NO + cex */
      Add $e \in \Sigma$ that witnesses the counterexample to $E$
    end
  end
end
```

by assigning the empty string $(\lambda)$ to both $S$ and $E$ (line 2). It then updates the observation $(S, E, T)$ by using membership queries (line 4). The observation table $(S, E, T)$ is made closed (from line 5 to line 8). When checking if $(S, E, T)$ is closed, all “?” results are considered as true, this is the same as the $L^*$–based assumption generation method shown in Sect. 3. If $(S, E, T)$ is closed, this means that we can create a candidate assumption from the closed observation table $(S, E, T)$, the algorithm then creates a candidate assumption $A_i$ represented by an LTS (from line 9 to line 10). We use $A_i, Q$ and $A_i, q_0$ to denote the set of states and the initial state of $A_i$, respectively. The idea of getting the following $tempQ$, comFinalState, and $C_{ikj}$ is shown in Fig. 4. Let $tempQ$ be the set of states of $A_i$ that does not include $A_i, q_0$ and $m$ be the number of states in $tempQ$ (line 11). The reason for creating the set $tempQ$ that does not include the initial state is that all candidate assumptions need the initial state. Therefore, we can try every possibility to remove other states, except the initial state. Later, the for loop from line 12 to line 28 is to consider all candidate assumptions that have the size $t + 1$ from 1 (i.e., the candidate assumption that has only one initial state) to $m + 1$ (i.e., the candidate assumptions that has the same set of states as $A_i$) (i.e., $0 \leq t \leq m$). With each value of $t$, the for loop from line 13 to line 27 is to check all candidate assumptions that have the same size of $t + 1$. The sets of states of these candidate assumptions are subsets of $tempQ \cup \{A_i, q_0\}$, where each subset is a $t$–combination of states from $tempQ$. Let $comFinalState$ be a $t$–combination of $m$ states of $tempQ$ (line 13). In the following steps (from line 14 to line 20), the algorithm constructs a candidate assumption, where $comFinalState \cup \{A_i, q_0\}$ be the set of states, from the current observation table. With this method of implementing, the algorithm considers all candidate assumptions that have the number of states from 1 to $m + 1$, i.e., from the candidate assumption that has only the initial state $A_i, q_0$ to all the candidate assumptions that have all $A_i, Q$ states. In line 14, the algorithm updates all “?” results in rows of $(S, E, T)$ which are not corresponding to any states in $comFinalState$ to true. The purpose of this step is to reduce the number of “?” results under consideration so that the number of $k$–combination of $n$ in the step below becomes smaller. Let $n$ be the number of “?” results remaining in the list after updating in line 14. For each $k$–combination of $n$ “?” results, the algorithm turns all those “?” results to false and the others “?” results to true (from line 17 to line 18). If the corresponding observation table $(S, E, T)$ is closed (line 19), the algorithm calculates a candidate $C_{ikj}$ (line 20) and asks the teacher an equivalence query (line 21), then stores result in result. An equivalence query
The table contains two properties: Key ∈ \{YES, NO, UNSAT\} (i.e., YES means the corresponding assumption satisfies the assume-guarantee rules; NO means the corresponding assumption does not satisfy assume-guarantee rules, however, at this point, we could not decide if the given system M does not satisfy p yet, we can use the corresponding counterexample cex to generate a new candidate assumption; UNSAT means the given system M does not satisfy p and the counterexample is cex); the other property is an assumption when Key is YES or a counterexample cex when Key is either NO or UNSAT. If result.Key is YES, the algorithm stops and returns the assumption associated with result (line 23). In this case, we have the locally minimum assumption generated. When the algorithm runs into line 29, it means that no smaller candidate assumption can be found in this iteration of the learning process, the algorithm asks an equivalence query for A_i (line 29). If the equivalence query result_equiResult.Key is YES, the algorithm stops and returns A_i as the needed assumption (line 31). If result_equiResult.Key is UNSAT, the algorithm returns UNSAT and the corresponding counterexample cex (line 33). This means that the given system M violates property p with the counterexample cex. Otherwise, the result_equiResult.Key is NO and a counterexample cex. The algorithm will analyze the counterexample cex to find a suitable suffix e. This suffix e must be such that adding it to E will cause the next assumption candidate to reflect the difference and keep the set of suffixes E closed.

The method to find e is not in the scope of this paper, please find more details in the improved L’ algorithm proposed by Rivest and Schapire [27]. The algorithm then adds it to E in order to have a better candidate assumption in the next iteration (line 35). The algorithm then continues the learning process again from line 4 until it reaches a conclusive result.

4.3 An Example

This section shows an example where the assumption generated by Algorithm 3 is smaller and stronger than the one generated by Algorithm 1. We cannot show that it is the locally minimum and strongest assumption in this example because that requires formal proof shown in Sect. 4.4 below. Given a system M = M_1 ∥ M_2 and a safety property p as shown in Fig. 5. This is actually the test case “TC6” in Table 1. In this figure, A is the assumption generated by Algorithm 1 and A_{LMS} is the assumption generated by Algorithm 3. Obviously, we have |A| = 6 > |A_{LMS}| = 3. In addition, we check if L(A_{LMS})_{Σ_m} ⊆ L(A) by using a common tool in the research community called LTSA [20], [21]. From formal proof about the correctness of Algorithm 3 in Sect. 4.4, we already have L(A_{LMS})_{Σ_m} ⊆ L(A). However, the reason for using another tool to check if L(A_{LMS})_{Σ_m} ⊆ L(A) is for double checking the result so that it becomes more reliable. For this purpose, we described A as a property and check if A_{LMS} ⊨ A using LTSA. The result is correct.
This means that $L(A_{LMS})_{\Sigma_1} \subseteq L(A)$. Consequently, $A_{LMS}$ is smaller, stronger and much simpler than $A$.

4.4 Correctness

For proving the correctness of the proposed algorithm, we follow the three steps to prove its soundness, completeness and termination. This proof is based on the correctness of the $L^*$-based assumption generation method presented in Sect. 3.

**Lemma 1.** (Soundness) Given a software $M$ consists of two components $M_1$ and $M_2$ (i.e., $M = M_1 \parallel M_2$) and a safety property $p$.

1. If Algorithm 3 returns a satisfied assumption $A$, then $M_1 \parallel M_2 \models p$ and $A$ is the satisfied assumption.
2. If Algorithm 3 reports “UNSAT + cex”, then $cex$ is the witness to $M_1 \parallel M_2 \models p$.

**Proof.**

1. In Algorithm 3, when the algorithm returns a satisfied assumption $A$ in either line 23 or line 31, it has already checked that the corresponding assumption (i.e., $C_{ik_j}$ or $A_j$) satisfies the assume-guarantee rules by asking the teacher an equivalence query and receiving the result of YES from the teacher (line 21 and line 29). Because we are using the same algorithm in resolving equivalence queries as the one that is being used in $L^*$-based assumption generation method in Sect. 3, by the algorithm’s correctness, we have $M_1 \parallel M_2 \models p$ and $A$ is the satisfied assumption.
2. In addition, when Algorithm 3 reports “UNSAT” and a counterexample $cex$, all of the candidate assumptions that have been asked to the teacher in line 21 did not satisfy the assume-guarantee rules. The equivalence query in line 29 has the result UNSAT and $cex$. When returning $UNSAT$ and $cex$, the teacher has checked that $M$ actually violates property $p$ and $cex$ is the witness. Therefore, thanks to the correctness of $L^*$-based assumption generation method presented in Sect. 3, $M_1\parallel M_2 \models p$ and $cex$ is the witness. □

**Lemma 2.** (Completeness). Let $M = M_1 \parallel M_2$ be an LTS that consists of two components $M_1$ and $M_2$ and $p$ be a safety property.

1. If $M_1 \parallel M_2 \models p$, then Algorithm 3 reports “YES” and the associated assumption $A$ is the required assumption.
2. If $M_1 \parallel M_2 \not\models p$, then Algorithm 3 reports “UNSAT” and the associated counterexample $cex$ is the witness to $M_1 \parallel M_2 \not\models p$.

**Proof.**

1. Compare Algorithm 1 and Algorithm 3, we can see that Algorithm 3 is different from Algorithm 1 at lines from 11 to 28. These steps are finite steps asking the teacher some more equivalence queries. Therefore, in the worst case, we cannot find out any satisfied assumption from these steps, the algorithm is equivalent to Algorithm 1. Therefore, if $M_1\parallel M_2 \models p$, then in the worst case, Algorithm 3 returns YES and the corresponding assumption $A$ thanks to the correctness of $L^*$-based assumption generation method shown in Sect. 3.

2. The same as the above description, in the worst case, where no satisfied assumption can be found in Algorithm 3 from line 11 to line 28, Algorithm 3 is equivalent to Algorithm 1. Therefore, if $M_1 \parallel M_2 \not\models p$, then Algorithm 3 will return UNSAT and the associated $cex$ is the counterexample thanks to the correctness of $L^*$-based assumption generation method shown in Sect. 3. □

**Lemma 3.** (Termination). Let $M = M_1 \parallel M_2$ be an LTS that consists of two components $M_1$ and $M_2$, and $p$ be a safety property. Algorithm 3 terminates in a finite number of learning steps.

**Proof.** The termination of Algorithm 3 follows directly from the above Lemma 1 and 2. □

**Lemma 4.** (Locally strongest assumption). Let $M = M_1 \parallel M_2$ be an LTS that consists of two components $M_1$ and $M_2$ and $p$ be a safety property. Assume that $M_1 \parallel M_2 \models p$ and Algorithm 3 does not return the assumption immediately after getting the first satisfied assumption (line 23 or line 31). It continues running to find all possible assumptions until all of the “?” results are turned into “true” results in the corresponding observation table. Let $\mathcal{W}$ be the subset of the set of those assumptions where they have $t$ states and $A \in \mathcal{W}$ be the first generated assumption in $\mathcal{W}$. $A$ is the locally strongest assumption in $\mathcal{W}$.

**Proof.** The key idea of Algorithm 3 is shown in Fig. 6. In this learning process, at the iteration $i^{th}$, we have a closed table $(S_i, E_i, T_i)$ and the corresponding candidate assumption $A_i$, in which all “?” results are considered as “true” (from line 5 to line 9). This means that all of the associated traces with

![Fig. 6](image-url)

The key idea of the locally minimum and strongest assumption generation method.
those "?" results are considered in the language of the assumption to be generated. Let \( m \) be the number of states of \( A_i \) that does not include \( q_0 \) (line 11). Let’s consider a set of assumptions that has \( t \) states, where \( 1 \leq t \leq m+1 \), that can be retrieved as follows (line 12). At first, we turn all "?" results corresponding to state \( q \notin tempQ \cup \{q_0\} \) to true (line 14). Then, if we have \( n "?" \) results left in \((S_i, E_i, T_i)\) (line 15), the algorithm will try to get \( k \)-combinations of \( n "?" \) results and consider all those "?" results as false (line 18), where \( k \) is from \( n \) to 1. This means that the algorithm will try to consider those corresponding traces as not in the language of the assumption to be generated. By doing this, the algorithm has tried every possibility that a trace does not belong to the language of the assumption to be generated. This is because \( k = n \) means no trace corresponding to "?" belongs to the language of the assumption to be generated. \( k \) is from \( n \) to 1 means only one trace corresponding to "?" results belongs to the language of the assumption to be generated, and so on. On the other hand, Algorithm 3 stops learning right after reaching a conclusive result. Therefore, in the worst case, where all of "?" results are considered as true, Algorithm 3 is equivalent to Algorithm 1. In other cases where there is a candidate assumption \( C_{ikj} \neq A_i \) that satisfies the assumption-guarantee rules, obviously, we have \( L(C_{ikj}) \subseteq L(A_i) \) because there are \( k "?" \) results in \((S_i, E_i, T_i)\) are considered as false. This means \( k \) traces that belong to \( L(A_i) \) but do not belong to \( L(C_{ikj}) \).

In case \( C_{ikj} \) exists, \( C_{ikj} \) is the locally strongest assumption because the algorithm has tried all possibilities that \( n, n-1, \ldots, k+1 \) traces which are corresponding to "?" results do not belong to the language of the assumption to be generated but it has not been successful yet. With this method of implementing, the algorithm has tried the strongest candidate assumption first, then weaker candidate assumptions later. On the other hand, with one value of \( k \), we have many \( k \)-combinations of \( n "?" \) results which can be considered as false. Each of such \( k \)-combination can be corresponding to one \( C_{ikj} \), where \( 1 \leq j \leq C_k^n \). Let \( L(C_{ikj}) \neq L(C_{ikl}) \) be two different candidate assumptions that are corresponding to two different \( k \)-combinations of \( n "?" \) results, where \( 1 \leq j \neq l \leq C_k^n \). It is a fact that in general, we cannot compare \( L(C_{ikj}) \) and \( L(C_{ikl}) \). Therefore, Algorithm 3 stops right after reaching the conclusive result and does not check all other \( C_{ikl} \)s with the same value of \( k \). As a result, the generated assumption must be the locally strongest assumption in the same iteration of the learning process.

We can remove line 9 from Algorithm 3. At that time, Algorithm 3 can generate stronger assumptions than those generated by Algorithm 1. However, it will not have the list of candidate assumptions of Algorithm 1 which plays a guideline role during the learning process. As a result, the algorithm will become much less efficient because of high time complexity.

**Lemma 5.** (Locally Minimum Assumption) Let \( M = M_1 \parallel M_2 \) be an LTS that consists of two components \( M_1 \) and \( M_2 \) and \( p \) be a safety property. Let’s assume that \( M_1 \parallel M_2 \models p \) and Algorithm 3 does not return the assumption immediately after getting the first satisfied assumption (line 23 or line 31). It continues running to find all possible assumptions until all of the "?" results are turned into "true" results in the corresponding observation table. Let \( \mathcal{A} \) be the set of those assumptions and \( A \) be the first generated assumption in \( \mathcal{A} \), A is the locally minimum assumption in \( \mathcal{A} \).

**Proof.** This proof comes directly from the proof of Lemma 4. From each baseline candidate assumption \( A_i \) at the learning iteration \( i^0 \), we start the checking process by considering all candidate assumptions which have the number of states from 1 (only have \( q_0 \)) to \( m+1 = |A_i| \) (from line 12 to line 13). Because the algorithm stops as soon as it reaches a conclusive result and with the way we find satisfied assumptions. As a result, the first found satisfied assumption must be the minimum one.

**Lemma 6.** (Complexity) Assume that Algorithm 1 takes \( M_{equivalence} \) equivalence queries and \( M_{mem} \) membership queries. Assume that at the iteration \( i^0 \), there are \( n_i "?" \) results. At each step \( t \) from 0 to \( n_i \), in the worst case where we have one candidate assumption for every \( k \)-combination of "?", it will takes \( \sum_{i=0}^{m} \sum_{l=1}^{n_i} C_m^l C_k^n \) equivalence queries, but no more membership queries. Therefore, in total and in the worst case, Algorithm 3 takes \( \sum_{i=1}^{M_{equivalence}} \sum_{l=1}^{n_i} C_m^l C_k^n \) equivalence queries and \( M_{mem} \) membership queries. As a result, the complexity of the proposed algorithm at iteration \( i^0 \) is \( O(2^m 2^n) \). For the target of reducing this complexity to a polynomial one, we have plan to another research that is based on the baseline candidate assumption \( A_i \), itself, not on its corresponding observation table \((S_i, E_i, T_i)\) anymore.

5. **Experiments**

In order to evaluate both Algorithm 1 and Algorithm 3, Sect. 5.1 shows some initial test cases with results. Later, Sect. 5.2 gives some discussions from several view points for both assumption generation methods.

5.1 **Experiment Results**

In order to evaluate the sizes and languages of the assumptions generated by Algorithm 3 in comparison with those of assumptions generated by Algorithm 1, we have implemented both Algorithm 1 and Algorithm 3 in a tool called Locally Minimum Assumption Generation Tool (LMAG). This tool is available at http://www.tranhoangviet.name.vn/p/lmag.html for reference. The tool is implemented using Microsoft Visual Studio 2017 Community. The test is carried out with some artificial test cases on a machine with the following system information: Processor: Intel(R) Core(TM) i5-3230M; CPU: @2.60GHz, 2601 Mhz, 2 Core(s), 4 Logical Processor(s); OS Name: Microsoft Windows 10 Enterprise. The experimental results are shown in Table 1. In this table, the following key indicators are used: the size of both software components and
property under testing ($|M_1|$, $|M_2|$, and $|p|$); the number of membership queries, equivalence queries used in both Algorithm 1 ($M_{AG}$, $EQ_{AG}$) and Algorithm 3 ($M_{LMAG}$, $EQ_{LMAG}$); the sizes of the generated assumptions by both algorithms ($|A_{AG}|$, and $|A_{LMAG}|$); language comparison is shown in column “Is stronger” (i.e., yes means $L(\text{LMAG}) < L(\text{AG})$ while no means $L(\text{LMAG}) \equiv L(\text{AG})$); and the time needed to generate assumptions for each test cases (AG Time (ms) and LMAG Time (ms)), respectively. From the above experimental results, we have the following observations:

- For some systems (test case 1, 2, and 4), Algorithm 3 can generate the same assumptions as the ones generated by Algorithm 1. For other systems (test case 3, 5, 6, 7, and 8), Algorithm 3 can generate smaller and stronger assumptions than the ones generated by Algorithm 1.
- Algorithm 3 requires more time to generate assumptions than Algorithm 1.
- The number of membership queries needed in Algorithm 3 is not always the same as the number of membership queries needed in Algorithm 1. This is because, in this case, we can find a satisfied locally minimum and strongest assumption at a step prior to the step where the original assumption generation method can generate the satisfied assumption.

The state space reduced by using the locally minimum and strongest assumptions generated by Algorithm 3 is shown in Table 2 for those test cases presented in Table 1. There are five indicators in Table 2 as follows: the size of the step 1 (denoted by $|S_{1_{ag}}| = |M_1| \times |A_{AG}| \times |p_{err}|$); the size of the step 2 of the assume-guarantee rules (denoted by $|S_{2_{ag}}| = |M_2| \times |A_{AG}|$) when doing assume-guarantee verification for $M \models p$ using the assumptions generated by Algorithm 1; the size of the step 1 (denoted by $|S_{1_{lmag}}| = |M_1| \times |A_{LMAG}| \times |p_{err}|$); the size of step 2 of the assume-guarantee rules (denoted by $|S_{2_{lmag}}| = |M_2| \times |A_{LMAG}|$) when doing assume-guarantee verification for $M \models p$ using the assumptions generated by Algorithm 3; how many percents of total state spaces we save when implementing assume-guarantee verification for $M \models p$ using the locally minimum and strongest assumptions generated by Algorithm 3 in comparison with that when using the assumptions generated by Algorithm 1 (denoted by $\% \downarrow = ( |S_{1_{ac}}| + |S_{2_{ac}}| ) - ( |S_{1_{lmag}}| + |S_{2_{lmag}}| ) ) / ( |S_{1_{ac}}| + |S_{2_{ac}}| )$). From Table 2, we can see that the locally strongest assumptions generated by Algorithm 3 can save up to 50% of the total state space for the verification of $M_1 \parallel M_2 \models p$. However, we have no obvious rule for how much we can save the state space of the problem $M \models p$ because that depends on each test case.

5.2 Discussions

In consideration about several aspects of Algorithm 1, Algorithm 3, the generated locally minimum and strongest assumptions, and the practical application of the proposed Algorithm 3, we have the following discussions.

Firstly, there are two advantages of Algorithm 3 in comparison with Algorithm 1 shown below.

- Assumptions generated by Algorithm 3 are smaller and stronger than those generated by Algorithm 1 as shown in Sect. 4.4.
- Although, time complexity of Algorithm 3 is higher than that of Algorithm 1 when generating the first assumption, the generated assumptions are used many times during software life cycle when verifying the evolving software as shown in the method proposed by Hung et al. [14, 16]. The more times we can reuse these assumptions, the more computational cost we save for software verification. Therefore, in the long run, the generated locally minimum and strongest assumptions will reduce effectively the verification cost of software. Further more, we are working on a method to reduce this time complexity of Algorithm 3.

Secondly, Algorithm 3 has two following disadvantages in comparison to Algorithm 1.

- Algorithm 3 needs to try every possible combinations of “?” results to see whether a trace can be in the language of $L(A)$, the complexity of the Algorithm 3 is clearly higher than that of Algorithm 1.
- Algorithm 1 is simpler than Algorithm 3. Therefore, it is easy for human to understand and faster than Algorithm 3 as shown in Table 1.

Thirdly, in regards to the importance of the locally minimum and strongest assumptions, there are several interesting points as follows.

- Modular verification for CBS is done by model checking the assume-guarantee rules with the generated assumption as one of its components. This is actually an emptiness problem of the languages of components of the system under checking and the assumption to be generated. For this reason, the computational cost of this checking is affected by the assumption size [29]. Therefore, the smaller assumption we have, the more reduction we gain for the computational cost of the verification.
- When a component is evolved after adapting some refinements in the context of software evolution, the

| No. | Test case | $|S_{1_{ac}}|$ | $|S_{2_{ac}}|$ | $|S_{1_{lmag}}|$ | $|S_{2_{lmag}}|$ | % ↓ |
|-----|-----------|---------------|---------------|---------------|---------------|-----|
| 1   | TC1       | 19            | 9             | 18            | 9             | 0%  |
| 2   | TC2       | 860           | 30            | 860           | 30            | 0%  |
| 3   | TC3       | 72            | 35            | 60            | 30            | 16% |
| 4   | TC4       | 36            | 15            | 36            | 15            | 0%  |
| 5   | TC5       | 60            | 20            | 45            | 16            | 24% |
| 6   | TC6       | 72            | 28            | 56            | 16            | 48% |
| 7   | TC7       | 432           | 28            | 216           | 16            | 50% |
| 8   | TC8       | 495           | 24            | 396           | 20            | 20% |
whole evolved CBS needs to be rechecked. In this case, we can reduce the cost of rechecking the evolved system by using the locally minimum and strongest assumptions [14, 16].

- Locally minimum and strongest assumptions mean less complex behavior (i.e., less number of states and/or transitions) so these assumptions are easier for human to understand. This is interesting for checking large-scale systems.

Last but not least, in consideration about the practical application of Algorithm 3, there are two important points about test cases shown in Table 1 as follows.

- The maximum size of test cases under checking is $TC2$ where $|M|\times|p_{err}| = |M_1|\times|M_2|\times|p_{err}| = 43\times5\times(3+1) = 860$. In the meantime, we only care about observable actions of the components. This allows us to apply the proposed method for practical systems that can have internally complex implementation, but have a limited number of observable actions such as the STS [25], and PLC [30]. Therefore, the experimental results can show reliable effectiveness when applying the proposed method and framework to large scale systems in practice.

- Although only some small test cases are carried out in the experiment, these test cases are able to show the fact that Algorithm 3 can generate smaller and stronger assumptions than those generated by Algorithm 1. As a result, the generated assumptions will reduce effectively the verification cost of CBS. On the other hand, we did not perform big experiments with systems in practice because that is another issue.

The key idea of this work is to consider that all possible combinations of traces which are not in the language of the assumption $A$ to be generated. This is done with the consideration of the size of $A$ from 1 to $m + 1$. We do that by considering from the case where no trace belongs to $L(A)$ to the case where all traces belong to $L(A)$. In addition, the algorithm terminates as soon as it reaches a conclusive result. As a result, the returned assumptions will be the locally minimum and strongest ones.

One important point when implementing Algorithm 3 is how to keep the observation table closed and consistent so that the language of the corresponding assumption candidate can be consistent with the observation table. This can be done with a suitable algorithm to choose suffix $e$ when adding a new item to the set of suffixes $E$ of the observation table in line 35. This algorithm is not in the scope of this paper. Details can be found in Rivest and Schapire’s paper [27].

The most complex step in Algorithm 3 is the step from line 12 to line 28 where the algorithm tries every possible size of candidate assumption $A_i$ from 1 state to $m + 1$ states together with $k$–combination of $n$ “?” results while considering them as false. Therefore, the complexity of Algorithm 3 depends on the number of “?” results in each steps of the learning process. For this reason, in Algorithm 3, we introduce an extra step in line 14 to reduce the number of “?” results that need to be processed. This is based on the following observation. For a closed table $(S, E, T)$, the corresponding DFA $D$ is created as defined in Definition 9, i.e., $D = (Q, \Sigma_M, \delta, q_0, F)$, where $Q = \{row(s) \mid s \in S\}$, $q_0 = row(A)$, $F = \{row(s) \mid s \in S \text{ and } T(s) = true\}$, $\Sigma_M = \Sigma$, and $\delta(row(s), a) = row(s.a)$. For a trace $\sigma \in \{row(s) = q \mid q \notin F\}$, $\sigma$ does not have much value since $q$ will be removed when generating the candidate assumption as defined in Remark 2. As a result, if a “?” result is corresponding to such a trace $\sigma$, we can update it to $true$ (line 14).

In general, Algorithm 3 does not always require more time to generate assumption than Algorithm 1. For example, if running Algorithm 1, it takes $M_{equi}$ steps to reach the satisfied assumption. However, there may be a step $i$ before $M_{equi}$ where a combination of “?” results considered as false results in a satisfied assumption. In this case, the time required to generate locally minimum assumption will be less than the time to generate assumption using Algorithm 1.

You may notice that Algorithm 3 bases on Algorithm 1 for making the observation table $(S, E, T)$ closed, creating candidate assumptions in the $k^{th}$ iteration of the learning process. We can apply the method that considers “?” results as false first when making the observation table $(S, E, T)$ closed, if the corresponding candidate assumption does not satisfy the assume-guarantee rules, we can go one step back to consider one by one “?” results as true until we find out the satisfied candidate assumption. However, this method of finding candidate assumptions has a very much greater time complexity. We chose the method that bases on Algorithm 1 as a framework for providing baseline candidate assumptions during the learning process. We only generate locally minimum and strongest candidate assumptions based on those baseline candidate assumptions. This method of learning can effectively generate locally minimum and strongest assumptions in an acceptable time complexity.

### 6. Related Works

There has been several existing works on improving the compositional verification for CBS that are relevant to our research [3], [8], [12], [15], [17], [18].

The framework proposed by Cobleigh et al. [8] can generate assumptions for compositional verification of CBS. However, because the algorithm is based on the language of the weakest assumption $(L(A_w))$, the generated assumptions are not minimum. With the focus on the size of generated assumptions, we improve the method so that the algorithm can generate locally minimum and strongest assumptions which can reduce the computational cost when verifying large-scale CBS.

Gupta et al. proposed a method to compute an exact minimal automaton to act as an intermediate assertion in assume-guarantee reasoning, using a sampling approach and a Boolean satisfiability solver [12]. This is an approach which is suitable to compute minimal separating assump-
tions for assume-guarantee reasoning for hardware verification. Our work focuses on generating the locally minimum and strongest assumptions when verifying CBS by improving the $L^*$-based assumption generation algorithm [8].

Hung et al. proposed a method for generating minimal assumptions, improving, and optimizing that method to generate those assumptions for compositional verification [15], [17], [18]. However, the proposed method based on search algorithm such as breadth-first search [15], depth-limited search [17], [18] in the space of closed observation tables to find the minimum satisfied assumptions. Our work shares the same observation that a trace $s$ that belongs to $L(A_w)$ does not always belong to the generated assumption language $L(A)$. In addition, the satisfiability problem is actually the language emptiness problem [29]. Our proposed method checks every combinations that a trace is not in the language of the assumption to be generated with the regards to its size from 1 to $m + 1$. Therefore, our work provides another way to generate the locally minimum assumptions. In the meantime, the generated assumptions by our proposed method are the locally strongest ones. These locally minimum and strongest assumptions can effectively reduce the computational cost when verifying CBS.

Chaki and Strichman proposed three optimizations to the $L^*$-based assumption generation method [3]. Among those three optimizations, the most important one is to develop a method for minimizing the alphabet used by the assumptions, which reduces the size of the assumptions and the number of queries required to construct them. However, the method does not generate the locally minimum and strongest assumptions as the proposed method in this paper.

7. Conclusion

We have presented a method to generate locally minimum and strongest assumptions for assume-guarantee verification of CBS. The method bases on a key observation that a trace $s$ may not be in the language of the generated assumption while the teacher answers true to the corresponding membership query of $s$. The paper then creates a variant of the existing membership queries answering technique where such true answers will be changed to “?”. The variant technique is integrated into an algorithm to generate locally minimum and strongest assumptions. The generated assumptions can be used effectively for verification of CBS, especially for large-scale ones.

Although experiments are carried out with small test cases, we are in progress of applying the method in systems in practice to verify its effectiveness. With large-scale systems in practice where verification cost becomes big issue, the locally minimum and strongest assumptions play an important role in reducing the verification cost. In general, there exist many methods that can generate locally minimum assumptions or generate locally strongest assumptions, we are working on a method that can generate either global minimum assumptions, or global strongest assumptions, or both minimum and strongest ones. Moreover, the current work is only for a software that contains only two components, we are also extending the method for software that consists of more than two components. In addition, this paper only considers checking a software against a safety property, we also have plan to extend the method for other kind of property like fairness and liveness properties, etc. Last but not least, we also have a lot of work to extend the method for more general system such as hardware, timed systems, and evolving ones.

Acknowledgments

This work is supported by the Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.03-2015.25.

References


