A Strongest Assumption Generation Method for Component-Based Software Verification

Hoang-Viet Tran[1], Chi-Luan Le[1,2], Pham Ngoc Hung[1]
[1] VNU University of Engineering and Technology
[2] University of Transport Technology
Email: 15028003@vnu.edu.vn, luanlc@utt.edu.vn, hungpn@vnu.edu.vn

***Abstract*—This paper presents a method for generating strongest assumptions during component-based software verification. The key idea of this method is to improve the technique for answering membership queries of the *Teacher* when responding to the *Learner* in the L^*–based learning process. This improved technique is then integrated into the breadth-first search algorithm in order to generate strongest assumptions. These assumptions will effectively reduce the computational cost when verifying component-based software, especially for large–scale and evolving ones. Some discussions about the importances of the generated strongest assumptions in modular verification are also presented in the paper.**

I. Introduction

The assume-guarantee reasoning [3] is a divide-and-conquer strategy to verify component-based software (CBS). This has been well known as a promising solution for the $state\ space\ explosion$ problem in formal verification. Let $M = M_1||M_2$ be a CBS and p be a required property. The goal of the verification is to check if M satisfies p without composing M_1 with M_2. In regards to assume-guarantee reasoning, finding an assumption A that satisfies both of the rules $\langle A\rangle\ M_1\ \langle p\rangle$ and $\langle true\rangle\ M_2\ \langle A\rangle$ is the key problem (i.e., to find an assumption A that $L(A||M_1)_{\uparrow\Sigma_p} \subseteq L(p)$ and $L(M_2)_{\uparrow\Sigma_A} \subseteq L(A)$) [3], where Σ_A and Σ_p are the sets of observable actions of A and p, respectively. Therefore, the stronger the assumption (i.e., an assumption with smaller language) is, the more computational cost can be reduced, especially when model checking large-scale and evolving CBSs. As a result, the strongest assumption (i.e., the assumption with smallest language) will be the best one to reduce computational cost when verifying CBSs. Although the works proposed in [5]–[7] can generate the minimal assumptions with the smallest number of states, the generated assumptions are not guaranteed to be strongest.

This paper presents a method to generate strongest assumptions for compositional verification. This method bases on an observation that the technique to answer membership queries from $Learner$ of $Teacher$ uses the weakest language, denoted by $L(A_W)$, to decide whether to return T or F to $Learner$ [3]. If a trace s belongs to $L(A_W)$, it returns T even if s may not belong to the language of the assumption to be generated. For this reason, the key idea of the proposed answering technique is that $Teacher$ will not directly return T to the query. It will return "?" to $Learner$ whenever the trace s belongs to $L(A_W)$. Otherwise, it will return F. After that, this technique is integrated into the breadth-first search algorithm for trying to add trace by trace from the "?" result list (first treated as F result) to the language of the assumption A to be generated. If the algorithm fails to find the assumption in one step, it will come back to one step before, try to change one "?" membership query result to T and continue the learning process. The algorithm terminates as soon as it reaches a conclusive result. Consequently, the generated assumptions, if exists, will be the strongest assumptions.

The rest of this paper is organized as follows. Section II presents background concepts used in this paper. Next, Section III reviews the original assumption generation method for compositional verification. After that, Section IV describes the proposed method to generate strongest assumptions. Related works to the paper are also analyzed in Section V. Finally, we conclude the paper in Section VI.

II. Background

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

LTSs. This paper uses *Labeled Transition Systems* (LTSs) to model behaviors of components. Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action unobservable to a component environment. We use π 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,
- $\Sigma \subseteq \mathcal{A}ct$ is a finite set of observable actions called the alphabet of M,
- $\delta \subseteq Q \times \Sigma \cup \{\tau\} \times Q$ is a transition relation, and
- $q_0 \in Q$ is the initial state.

Traces. A trace σ of an LTS M is a sequence of observable actions that M can perform starting at its initial state.

Definition 2: (Trace). A trace σ of an LTS $M = \langle Q, \Sigma, \delta, q_0\rangle$ is a finite sequence of actions $a_1a_2...a_n$, such that there exists a sequence of states starting at the initial state (i.e., $q_0q_1...q_n$) such that for $1 \leq i \leq n$, $(q_{i-1}, a_i, q_i) \in \delta$, $q_i \in Q$.

Definition 3: (Concatenation operator). Given two sets of event sequences P and Q, $P.Q = \{pq \mid p \in P, q \in Q\}$, where pq presents the concatenation of the event sequences p and q.

Note 1: The set of all traces of M is called the language of M, denoted by $L(M)$. Let $\sigma = a_1a_2...a_n$ 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 $Q = \{q_0, q_1, ..., q_n\}$, and $\delta = \{(q_{i-1}, a_i, q_i)\}$, where $1 \leq i \leq n$.

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

Definition 4: (Parallel composition operator). The parallel composition between $M_1 = \langle Q_1, \Sigma_{M_1}, \delta_1, q_0^1 \rangle$ and $M_2 = \langle Q_2, \Sigma_{M_2}, \delta_2, q_0^2 \rangle$, denoted by $M_1 \| M_2$, is defined as follows. If $M_1 = \prod$ or $M_2 = \prod$, then $M_1 \| M_2 = \prod$, where $\prod$ denotes the LTS $\langle \{\pi\}, Act, \emptyset, \pi \rangle$. Otherwise, $M_1 \| M_2$ is an LTS $M = \langle Q, \Sigma, \delta, q_0 \rangle$ where $Q = Q_1 \times Q_2$, $\Sigma = \Sigma_{M_1} \cup \Sigma_{M_2}$, $q_0 = (q_0^1, q_0^2)$, and the transition relation δ is given by the following rules:

$$(i) \frac{\alpha \in \Sigma_{M_1} \cap \Sigma_{M_2}, (p, \alpha, p') \in \delta_1, (q, \alpha, q') \in \delta_2}{((p, q), \alpha, (p', q')) \in \delta} \quad (1)$$

$$(ii) \frac{\alpha \in \Sigma_{M_1} \backslash \Sigma_{M_2}, (p, \alpha, p') \in \delta_1}{((p, q), \alpha, (p', q)) \in \delta} \quad (2)$$

$$(iii) \frac{\alpha \in \Sigma_{M_2} \backslash \Sigma_{M_1}, (q, \alpha, q') \in \delta_2}{((p, q), \alpha, (p, q')) \in \delta} \quad (3)$$

Safety LTSs, Safety Property, Satisfiability and Error LTSs.

Definition 5: (Safety LTS). A safety LTS is a deterministic LTS that contains no π states.

Note 2: A safety property asserts that nothing bad happens for all time. The 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 Σ_p.

Definition 6: (Satisfaction Relation). an LTS M satisfies p, denoted by $M \models p$, if and only if $\forall \sigma \in L(M)$: $(\sigma_{\uparrow \Sigma_p}) \in L(p)$, where $\sigma_{\uparrow \Sigma_p}$ denotes the trace obtained by removing from σ all occurrences of actions $a \notin \Sigma_p$.

Note 3: 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 π state. p_{err} is defined as follows:

Definition 7: (Error LTS). An *error LTS* of a property $p = \langle Q, \Sigma_p, \delta, q_0 \rangle$ is $p_{err} = \langle Q \cup \{\pi\}, \Sigma_p, \delta', q_0 \rangle$, where $\delta' = \delta \cup \{(q, a, \pi) \mid a \in \Sigma_p \text{ and } \nexists q' \in Q : (q, a, q') \in \delta\}$.

Remark 1: The error LTS is 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 $M \| p_{err}$ is then computed. If the state π is reachable in the compositional system then M violates p. Otherwise, it satisfies p.

Assume-Guarantee Reasoning. An assume-guarantee rule is defined as follows.

Definition 8: (Compositional Verification). Let M be a system which consists of two components M_1 and M_2, p be a property, and A be an assumption about M_1's environment. The compositional verification rule is described as following formula [3].

$$\frac{(step\ 1)\ \ \langle A \rangle\ M_1\ \langle p \rangle \qquad (step\ 2)\ \ \langle true \rangle\ M_2\ \langle A \rangle}{\langle true \rangle\ M_1 || M_2\ \langle p \rangle}$$

Note 4: We use the formula $\langle true \rangle\ M\ \langle A \rangle$ to represent the LTS $M \| A_{err}$. The formula $\langle A \rangle\ M\ \langle p \rangle$ 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 LTS $A \| M \| p_{err}$ and check if the error state π is reachable. If it is, then the formula is violated, otherwise it is satisfied.

Note 5: (Weakest Assumption). Weakest assumption A_W describes exactly those traces over the alphabet $\Sigma = (\Sigma_{M_1} \cup \Sigma_p) \cap \Sigma_{M_2}$ which, the error state π is not reachable in the compositional system $M_1 \| p_{err}$. The weakest assumption A_W means that for any environment component E, $M_1 \| E \models p$ if and only if $E \models A_W$.

Note 6: (Strongest Assumption). Let A_S be assumptions that satisfies the compositional verification rule in Definition 8. If for all A satisfying the compositional verification rule in Definition 8: $L(A_S) \subseteq L(A)$. We call A_S the $strongest\ assumption$.

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

- $S \in \Sigma^*$ is a set of prefixes,
- $E \in \Sigma^*$ is a set of suffixes, and
- $T : (S \cup S.\Sigma).E \rightarrow \{T, F\}$. With a string $s \in \Sigma^*$, $T(s) = T$ means $s \in L(A)$, otherwise $s \notin L(A)$.

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 of $S.\Sigma$ has a matching row s' in S.

III. Original Assumption Generation Method

A. The L^* Algorithm

L^* algorithm [1] is an incremental learning algorithm that is developed by Angluin and later improved by Rivest and Schapire [8]. L^* can learn an unknown regular language and generate a deterministic finite automata (DFA) that accepts it. Let U be an unknown regular language over some alphabet Σ. L^* will produce a DFA M such that $L(M) = U$. In this learning model, the learning process is performed by the interaction between $Learner$ (i.e., L^*) and $Teacher$. The interaction is shown in Figure 1. The $Teacher$ is an object that must be able to answer the following two types of queries from $Learner$.

- Membership queries: These queries consist of a string $\sigma \in \Sigma^*$ (i.e., "is $\sigma \in U$?"). The answer is T if $\sigma \in U$, and F otherwise.
- Equivalence queries: These queries consist of a candidate DFA M whose language the algorithm believes to be identical to U ("is $L(M) = U$?"). The answer is YES if $L(M) = U$. Otherwise $Teacher$ returns NO and a

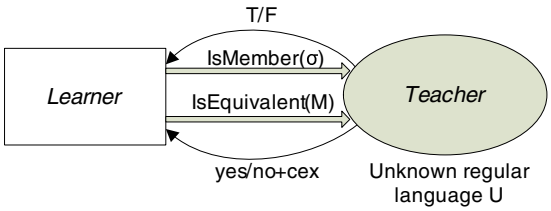

Fig. 1. The interaction between L* $Learner$ and the $Teacher$.

counterexample cex which is a string in the symmetric difference of $L(M)$ and U.

B. Generating Assumptions using L^* Algorithm

Given a CBS M that consists of two components M_1 and M_2 and a property p. The original assumption learning algorithm proposed in [3] generates a contextual assumption using the L^* algorithm [1]. The details of this algorithm is shown in Algorithm 1. In order to learn an assumption A,

Algorithm 1: Learning assumptions for compositional verification

```
1  begin
2      Let S = E = {λ}
3      while true do
4          Update T using membership queries
5          while (S, E, T) is not closed do
6              Add sa to S to make (S, E, T) closed where
               s ∈ S and a ∈ Σ
7              Update T using membership queries
8          end
9          Construct candidate DFA M from (S, E, T)
10         Make the conjecture C from M
11         Ask equivalence query for the conjecture C
12         if C is correct then
13             return C
14         else
15             Add e ∈ Σ* that witness the counterexample
               to E
16         end
17     end
18 end
```

Algorithm 1 maintains an observation table (S, E, T). The algorithm starts by initializing S and E with the empty string λ (line 2). After that, the algorithm updates (S, E, T) by using membership queries (line 4). While the observation table is not closed, the algorithm continues adding sa 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 $Teacher$ (from line 9 to line 11). If C is the needed assumption, the algorithm stops and returns C (line 13). Otherwise, it analyzes the returned counterexample cex to find a suitable suffix e. It then adds e to E (line 15) and continues the learning process again from line 4. The incremental composition verification during the iteration i is shown in Figure 2.

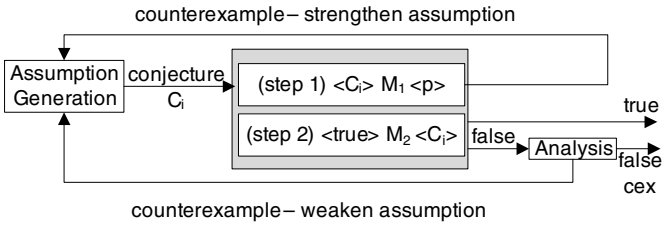

Fig. 2. Incremental compositional verification during iteration i.

IV. Learning Strongest Assumptions using Backtracking Algorithm

A. An Improvement to the Technique for Answering Membership Query

In Algorithm 1, $Learner$ updates the observation table during the learning process by asking $Teacher$ a membership query if a trace s belongs to the language of an assumption A that satisfies the assume-guarantee rule (i.e., $s \in L(A)?$). In

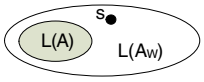

Fig. 3. The relationship between $L(A)$ and $L(A_W)$.

order to answer this query, the algorithm in [3] bases on the language of *weakest assumption* ($L(A_W)$) to consider if the given trace belongs to $L(A)$. If $s \in L(A_W)$, the algorithm re-

Algorithm 2: Improved membership query answering algorithm

input : A trace $s = a_0 a_1 ... a_n$
output: If $s \in L(A_W)$ then "?", otherwise F

```
1 begin
2     if ⟨[s]⟩M1⟨P⟩ then
3         return "?"
4     else
5         return F
6     end
7 end
```

turns T, otherwise, it returns F. However, when the algorithm returns T, it has not known whether s really belongs to $L(A)$. This is because for all A satisfying the compositional rule in Definition 8: $L(A) \subseteq L(A_W)$. The relationship between $L(A)$ and $L(A_W)$ is shown in Figure 3. For this reason, we propose an improvement to the membership query answering technique described in Algorithm 2. In this algorithm when $Teacher$ receives a membership query for a trace $s = a_0 a_1 ... a_n \in \Sigma^*$, it first builds an LTS $[s]$. It then model checks $\langle [s] \rangle M_1 \langle P \rangle$. If $true$ is returned (i.e., $s \in L(A_W)$), $Teacher$ returns "?"

(line 3). Otherwise, $Teacher$ returns F (line 5). The "?" result is then used in $Learner$ to learn the strongest assumption.

B. Generating the Strongest Assumption

In order to employ the improved technique for answering membership queries proposed in Algorithm 2 to generate assumption while doing component-based software verification, we use the breadth-first search algorithm shown in Algorithm 3. The key idea of this algorithm is that $Learner$ maintains a list of checkpoints ($CPList$) during the assumption learning process. Each of the checkpoints in the list is a couple of an observation table and its corresponding membership query results list $\langle OT, MQResult\rangle$. Whenever there is a change in the observation table (i.e., adding a new state to S or a suffix to E), the algorithm creates a new checkpoint and add it to the end of the list. All of the "?" in $MQResult$ of the previous checkpoint must be turned to F before using them in $MQResult$ of the new checkpoint because dealing with this new checkpoint, those "?" results are equivalently considered as F. At each of the learning steps, $Learner$ updates $CPList[0].OT$ by calling Algorithm 4 with input $CPList[0]$ (line 4). It then tries to create a conjecture and analyze an equivalence query result using $CPList[0]$ by calling Algorithm 5 and put the result into $result$ (line 5). In case $result$ is YES, Algorithm 3 returns the assumption associated with $result$ (line 8). In case $result$ is $UNSAT$ (unsatisfied) and a counterexample cex, the algorithm checks if there is any "?" result in $CPList[0].MQResult$. If there is, it turns the last "?" in that list to T. The observation table now may contain duplicate lines in S, the algorithm must remove these lines and keep only one of them in S (lines from 10 to 12). If there is no "?" result in $CPList[0].MQResult$, the algorithm removes the first checkpoint from $CPList$ (line 14). If there is no checkpoint left in $CPList$, this means that the given system really violates the property, the algorithm returns $UNSAT$ and cex is the counterexample (line 16). If $CPList$ is not empty, the algorithm comes back to line 3 and continues the learning process using the first checkpoint in $CPList$. In case $result$ is $NO + cex$ or ADD_STATE, this is the case where the observation table in the current checkpoint should be changed by adding a new suffix to E or a new state to S, the algorithm changes all of the "?" results in the new checkpoint associated with $result$ to F and add that checkpoint to the end of $CPList$ (lines from 20 to 21). The algorithm then checks $CPList[0].MQResult$ in order to update its last "?" result to T or remove $CPList[0]$ from the list (lines from 22 to 26). After that, it comes back to line 3 and continues the learning process using the first checkpoint in $CPList$.

C. Improving the Technique for Updating Observation Table

In Algorithm 3, each learning step is started by updating the observation table in the current checkpoint (line 4 in Algorithm 3). The corresponding $MQResult$ is also updated during the update process. Details of the algorithm is shown in Algorithm 4. The algorithm accepts a checkpoint CP as an input parameter. Each item in $CP.MQResult$ is in form of a couple $\langle s, t\rangle$, where s is a trace in the corresponding membership query asked to $Teacher$ and $t \in \{T, F, ?\}$. For each string str to be passed to $Teacher$ in a membership query, if str is in $CP.MQResult$, the corresponding result from $CP.MQResult$ is retrieved and stored in t. The algorithm treats all "?" result as F by updating the corresponding T in $CP.OT$ with F whenever t equals to "?" or F (lines from 7 to 8). Otherwise, it updates the corresponding T in $CP.OT$ with T (line 10). In case str is not in $CP.MQResult$, the algorithm asks a membership query for str (line 13) and stores the result in $CP.MQResult$ (line 14) and then update the corresponding T in $CP.OT$ as the above case (lines from 15

Algorithm 3: Learning strongest assumptions algorithm

```
1  begin
2      Let S = E = {λ}, CPList ← {⟨(S, E, T), empty⟩}.
3      while CPList ≠ ∅ do
           /* call Algorithm 4(CPList[0])      */
4          Update CPList[0].OT.
           /* call Algorithm 5(CPList[0])      */
5          result ← analyze equivalence query result.
6          switch result do
7              case YES
8                  return yes and result.assumption.
9              case UNSAT
10                 if CPList[0].MQResult contains "?"
                   result then
11                     Turn the last "?" result of
                       CPList[0].MQResult to T.
12                     Remove duplicate lines in
                       CPList[0].OT.S.
13                 else
14                     Remove the 0^th item from CPList.
15                     if CPList = ∅ then
16                         return ⟨UNSAT, result.cex⟩.
17                     end
18                 end
19             case NO + cex or ADD_STATE
20                 Turn all "?" results of
                   result.CP.MQResult to F.
21                 Add result.CP to the end of CPList.
22                 if CPList[0].MQResult contains "?"
                   result then
23                     Turn the last "?" result of
                       CPList[0].MQResult to T.
24                     Remove duplicate lines in
                       CPList[0].OT.S.
25                 else
26                     Remove the first item in CPList.
27                 end
28             end
29         endsw
30     end
31 end
```

Algorithm 4: The improved algorithm for updating observation table

input : A checkpoint CP
output: The checkpoint CP with both OT and $MQResult$ updated

```
1  begin
2      forall the s or sa ∈ CP.OT.S do
3          forall the e ∈ E do
4              str ← s.e or sa.e
5              if CP.MQResult contains str then
6                  t ← get value of str from CP.MQResult
7                  if t = "?" or t = F then
8                      Update the corresponding T in CP.OT with F
9                  else
10                     Update the corresponding T in CP.OT with T
11                 end
12             else
13                 t ← ask membership query for str
14                 Store ⟨str, t⟩ to CP.MQResult
15                 if t = "?" or t = F then
16                     Update the corresponding T in CP.OT with F
17                 else
18                     Update the corresponding T in CP.OT with T
19                 end
20             end
21         end
22     end
23     return CP.
24 end
```

Algorithm 5: The analysis for equivalence query result

input : A checkpoint CP

```
1  begin
2      IsClosed, s, a ← check if CP.OT is closed.
3      if IsClosed = false then
4          CP.OT.S ← s.a.
5          return ⟨ADD_STATE, CP⟩.
6      end
7      if IsClosed = true then
8          conjecture ← create conjecture from CP.OT.
9          result ← ask equivalence query for conjecture.
10         if result = YES then
11             return ⟨YES, result.assumption⟩.
12         else if result = NO + cex then
13             suffix ← find suitable suffix from cex.
14             if a suffix exists then
15                 Add suffix to CP.OT.E.
16                 return ⟨NO + cex, CP⟩.
17             else
18                 return ⟨UNSAT, cex⟩.
19             end
20         else if result = UNSAT + cex then
21             return ⟨UNSAT, cex⟩.
22         end
23     end
24 end
```

to 18). After updating $CP.OT$, the checkpoint CP is returned (line 23).

D. Analyzing the Equivalence Query Result

After updating observation table using Algorithm 4, Algorithm 3 tries to create a candidate assumption and asks an equivalence query by calling Algorithm 5 (line 5 in Algorithm 3). Details of the process is described in Algorithm 5. The algorithm accepts a checkpoint CP as the input. It starts by checking if $CP.OT$ is closed. If it is not closed, a new state is added to $CP.OT.S$ and the checkpoint is returned with the result of ADD_STATE (lines from 3 to 5). If it is closed, the corresponding conjecture is created from $CP.OT$ (line 8). The algorithm then uses this conjecture to ask $Teacher$ equivalence query and stores the result in $result$ (line 9). If $result$ is YES, the algorithm returns YES and the associated assumption (line 11). If $result$ is NO and a counterexample cex, the algorithm analyzes the returned cex to find an appropriate $suffix$ to be added to $CP.OT.E$ (line 13). If such $suffix$ exists, the algorithm adds the $suffix$ to $CP.OT.E$ then returns $NO + cex$ and the corresponding checkpoint CP (lines from 14 to 16). If such $suffix$ does not exist, the algorithm returns $UNSAT$ with cex as the corresponding counterexample (line 18). If $result$ is $UNSAT$ and a counterexample cex, the algorithm also returns $UNSAT$ with cex as the associated counterexample (line 21).

E. Discussion

In regards to the importances of the generated strongest assumptions when verifying component-based software, there are several interesting points as follows:

- Modular verification for component-based software is done by model checking the assume-guarantee rule with the generated assumption as one of its components. This is actually the emptiness problem of the intersection of the language of components of the system under checking and the complement of the language of the assumption to be generated. For this reason, the computational cost of this checking is affected by the assumption language. Therefore, the stronger assumption we have, the more reduction we gain for the computational cost of the verification.
- The key idea of this work is to assume that all of the possible traces are not in the language of the assumption A to be generated. Then, we add one by one trace to $L(A)$ until we have the needed assumptions. Besides, the algorithm terminates as soon as it reaches a conclusive

result. Because of this, the returned assumption must be the strongest one.
- When a component is evolved after adapting some refinements in the context of software evolution, the whole evolved CBS needs to be rechecked. In this case, we can reduce the cost of rechecking the evolved system by using the strongest assumption.
- Strongest assumption means less complex behavior so this assumption is easier for human to understand. This is interesting for checking large–scale systems.

Despite the advantages mentioned above, the algorithm needs to try every possible trace to see if it can be in the language of $L(A)$, the complexity of the Algorithm 3 is clearly higher than the complexity of Algorithm 1 proposed in [3].

V. Related Works

There are many researches related to compositional verification for component-based software. Consider only the most current works, we can refer to [2]–[7].

The framework proposed in [3] by Cobleigh et al. can generate assumptions for compositional verification of CBS. However, because the algorithm based on $L(A_W)$ to generate assumptions, the generated assumptions are not strongest. By observing this, we focus on improving the method so that the algorithm can generate strongest assumptions which can reduce the computational cost when verifying large–scale CBSs.

In [4], 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. This is an approach which is suitable to compute minimal separating assumptions for assume-guarantee reasoning for hardware verification. Our work focuses on generating the strongest assumption when verifying component-based software by improving the original algorithm proposed in [3].

In a series of papers of [5]–[7], Hung et al. proposed a method for generating minimal assumptions, improving, and optimizing that method to generate those assumptions for compositional verification. However, the generated minimal assumptions in these works mean to have a minimal number of states. 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 $L(A)$. Besides, the satisfiability problem is actually the emptiness problem of languages. Therefore, our work will effectively reduce the computational cost when verifying component-based software.

Chaki and Strichman proposed three optimizations in [2] to the L^*–based automated assume-guarantee reasoning algorithm for the compositional verification of concurrent systems. 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 strongest assumptions as the proposed method in this paper.

VI. Conclusion

We have presented a method to generate strongest assumptions for assume-guarantee verification of component-base software. The key idea of this is to use the breadth-first search algorithm to try to add one by one trace from the weakest assumption in order to generate candidate assumptions. Because the algorithm terminates as soon as it reaches the conclusive result, the generated assumptions are the strongest. These assumptions can effectively reduce the computation cost when doing verification for CBSs, especially for large-scale and evolving ones.

Although the proposed method can logically generate strongest assumptions for compositional verification, we will need to prove formally its soundness, completeness, and termination in our future works. Besides, we are in progress of applying the proposed method for software in practice to prove its effectiveness. Moreover, we are investigating how to generalize the method for larger systems, i.e., systems contain more than two components. Besides, the current work is only for safety properties, we are going to extend our proposed method for checking other properties such as liveness properties and apply the proposed method for general systems, e.g., hardware systems, real-time systems.

Acknowledgments

This work is supported by the project no. QG.16.31 granted by Vietnam National University, Hanoi (VNU).

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, Nov. 1987.

[2] S. Chaki and O. Strichman. *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007. Proceedings*, chapter Optimized L*-Based Assume-Guarantee Reasoning, pages 276–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[3] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, pages 331–346, Berlin, Heidelberg, 2003. Springer-Verlag.

[4] A. Gupta, K. L. Mcmillan, and Z. Fu. Automated assumption generation for compositional verification. *Form. Methods Syst. Des.*, 32(3):285–301, June 2008.

[5] P. N. Hung, V.-H. Nguyen, T. Aoki, and T. Katayama. An improvement of minimized assumption generation method for component-based software verification. In *Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), 2012 IEEE RIVF International Conference on*, pages 1–6, Feb 2012.

[6] P. N. Hung, V. H. Nguyen, T. Aoki, and T. Katayama. On optimization of minimized assumption generation method for component-based software verification. *IEICE Transactions*, 95-A(9):1451–1460, 2012.

[7] P. Ngoc Hung, T. Aoki, and T. Katayama. *Theoretical Aspects of Computing - ICTAC 2009: 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, chapter A Minimized Assumption Generation Method for Component-Based Software Verification, pages 277–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[8] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 411–420, New York, NY, USA, 1989. ACM.