On Implementation of the Assumption Generation Method for Component-Based Software Verification

Chi-Luan Le[1,2], Hoang-Viet Tran[2], Pham Ngoc Hung[2],
luanlc@utt.edu.vn, {15028003, hungpn}@vnu.edu.vn

[1] University of Transport Technology
[2]VNU University of Engineering and Technology

Abstract. The assume-guarantee verification has been recognized as a promising method for solving the *state space explosion* in modular model checking of component-based software. However, the counterexample analysis technique used in this method has huge complexity and the computational cost for generating assumptions is very high. As a result, the method is difficult to be applied in practice. Therefore, this paper presents two improvements of the assume-guarantee verification method in order to solve the above problems. The first one is a counterexample analysis method that is simple to implement but effective enough to prevent the verification process from infinite loops when considering the last action of *counterexample* as suffix in implementation. This is done by finding a suffix that can make the observation table not closed when being added to the suffix set of the table and use that suffix for the learning process. The second one is a reduction of the number of membership queries to be asked to *teacher* when learning assumptions. This results in a significantly faster speed in generating assumption than that of the original algorithm. An implemented tool and experimental results are also described to show the effectiveness of the improvements.

1 Introduction

Software quality nowadays plays an important role in our society because software has helped us to improve all matters of our life such as our homes, schools, jobs, etc. With almost all of software in practice, testing has been considered as a major solution for guaranteeing software quality. However, testing is not enough for high quality software that requires no error such as plane, train controller systems, etc. With such systems, we will need formal methods to ensure the correctness of systems in both of design and implementation phases. Therefore, many researches have been carried out to improve software quality while keeping software development fast and effective. Two well known approaches that address this problem are theorem proving and model checking [4]. In regards to model checking, assume-guarantee verification has been used as one of the most important method to verify component-based software (CBS) with model checking. This is because it helps us to do verification in a full automatic manner. Moreover, the method is not only suitable for CBS but also for solving the *state space explosion* problem in model checking. It does this by allowing us to verify a target system composed from components by model checking each of them separately.

In assume-guarantee verification, the major problem is how to generate assumptions that satisfy the rules of assume-guarantee. The problem can be solved by using the proposed framework in [5]. The idea of the framework is to generate an assumption A as a contextual assumption about system environment using the L^* algorithm [1,12]. In this framework, the learning process is performed by the interaction between L^* (from now on called *learner*) and *teacher*. During the learning process, the *teacher* must be able to answer correctly two kinds of queries from *learner* to learn the unknown regular language of A, denoted by $L(A)$. The first one is membership query which is to ask whether a trace σ belongs to $L(A)$. The second one is equivalence query which is to ask if the language of a conjecture C (denoted by $L(C)$) is equivalent to $L(A)$. If $L(C)$ is equivalent to $L(A)$, then C is the needed assumption and *teacher* answers *yes* to *learner*. If $L(C)$ is not equivalent to $L(A)$ but the system does not really violate the given property, *teacher* will return a counterexample *cex* that witnesses the difference between $L(C)$ and $L(A)$. Otherwise, *teacher* will return *no and cex*, where *cex* is corresponding to the actual violation. For the purpose of generating conjectures, *learner* maintains an observation table in form of (S, E, T) and updates it frequently by using membership queries. Whenever this table is closed, *learner* will create a conjecture from the table and submit it to *teacher* as an equivalence query. When *teacher* returns *cex*, *learner* analyzes it to find out a suffix that should be added to E for generating a better conjecture. The algorithm proposed in [5], instead of providing a detailed method to retrieve the suffix to be added to E, refers to the method proposed in [12] for retrieving that suffix. However, the method proposed in [12] has huge complexity. Implementing this method will not always suitable for large scale systems in practice. On the other hand, if we simply add the last action of *cex* to E as suffix, it will lead to a case where the learning process comes into an infinite loop. Therefore, the process fail to generate assumption even though the system does not violate the given property. Moreover, although the assume-guarantee verification method have been well known for a long time, its application in practice is very limited due to the high computational cost in generating assumptions. This is due to the reason that there are many duplicate membership queries which have been asked to *teacher*. Therefore, the method needs improvements so that it can run correctly with lower cost to generate assumptions.

This paper proposes two improvements of the assumption generation method. The first one is an algorithm that simplifies the counterexample analysis process so that it can run without infinite loop in most of the cases in a reasonable time cost. The key idea of this algorithm is to try to add each of the suffixes with the length from one to the length of *cex* to E. After that, the table is updated to see if the updated one is closed. If a suffix can make the observation table not closed, we can add it to the suffix list E. The table is then used to generate a new conjecture to submit to *teacher* as an equivalence query. The learning process continues until *teacher* answers *yes* or *no* with *cex*. The second one is an algorithm to reduce the number of membership queries when updating observation tables. The key idea of the algorithm is that we should only ask membership query for a specific trace σ only once and store the result in a dictionary for later using in the learning process. As a result, the number of membership queries to be submitted to *teacher* will be minimal. This results in the reduction of the computational cost for generating assumptions.

The rest of this paper is organized as follows. At first, we review the original assumption generation method in Section 2. An improved counterexample analysis algorithm will be presented in Section 3. This section will also describe the algorithm to reduce the number of membership queries when learning assumptions. A support tool and experimental results will be shown in Section 4. Section 5 presents an overview about the researches that are related to the topic. Finally, we conclude the paper in Section 6.

2 The Original Assumption Learning Algorithm

2.1 Generating Assumption using L^* Algorithm

Given a system M that consists of two components M_1 and M_2 and a property p. The original assumption learning algorithm proposed in [5] generates a contextual assumption using the L^* algorithm [1]. The details of this algorithm are

Algorithm 1: Learning Assumptions for Compositional Verification

```
1  begin
2      Let $S = E = \{\lambda\}$
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 $s \in S$ and $a \in \Sigma$
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 $e \in \Sigma^*$ that witnesses the counterexample to $E$
16         end
17     end
18 end
```

shown in Algorithm 1. In order to learn assumption A, Algorithm 1 maintains an observation table (S, E, T). The algorithm starts by initializing S and E with λ (i.e., an empty string) (line 2). After that, the algorithm updates the observation table (S, E, T) by using membership queries (line 4). While (S, E, T) 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 (S, E, T) and asks *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 add the suffix e that witnesses the counterexample to E (line 15) and continues the learning process again from line 4.

2.2 Updating Observation Table

While Algorithm 1 is learning assumption, a very important step is to update the observation table. The details of this step are presented in Algorithm 2. For

Algorithm 2: Updating Observation Table

```
input  : An observation table $(S, E, T)$
output: The updated observation table $(S, E, T)$
1 begin
2     forall the $s \in S$ or $sa \in S$ do
3         forall the $e \in E$ do
4             $t \leftarrow$ ask membership query for $s.e$ or $sa.e$
5             Update the corresponding $T$ in $(S, E, T)$ with $t$
6         end
7     end
8     return $(S, E, T)$
9 end
```

every $s \in S$ or $sa \in S.\Sigma$ (line 2), the algorithm concatenates this with each of e in E (line 3). Then, it asks *teacher* a *membership query* for $s.e$ or $sa.e$ (line 4) (where "." is the concatenation operator). After that, it updates the corresponding T in (S, E, T) with the result of this membership query (line 5). When finishing this process, it returns the updated observation table (S, E, T) (line 8).

3 Two Improvements for the Assumption Generation Method

From our observation, we have seen that if we simply apply Algorithm 2 to update observation tables, there will be a lot of duplicated membership queries.

This will dramatically affect the assumption generation process when dealing with large scale systems. Following sub-sections propose algorithms to choose suffix from counterexample and to reduce the number of membership queries to be asked to *teacher*.

3.1 An Improvement on Counterexample Analysis

Although the idea of trying all of the possible suffixes with the length increased one by one from the counterexample is not new and can be found in other works such as in [9], no one has ever applied the idea in assume-guarantee reasoning. When applying the idea in assume-guarantee reasoning, we have an effective method to analyze the counterexample as shown in Algorithm 3. When *teacher*

Algorithm 3: Choosing suffix from counterexample

```
   input  : The current observation table $(S, E, T)$, the counterexample $cex$
   output: $yes + e$ or $no$
1  begin
2      foreach $counter = 1$ to $cex$'s length do
3          $OT \leftarrow$ the cloned table of $(S, E, T)$
4          $e \leftarrow suffix$ with length is counter
5          Try to add $e$ to $OT's\ E$
6          Update $OT$ with membership queries
7          if $OT$ is not closed then
8              return $yes + e$
9          end
10     end
11     return $no$
12 end
```

processes an equivalence query with a conjecture argument C, if C is not the satisfied assumption, but M does not violate the property p, *teacher* will return a counterexample cex. Algorithm 3 analyzes cex to choose an appropriate suffix to add to E. The idea of this algorithm is to try to add each of the suffix which has length from one to cex's length to E to find out which suffix will make the observation table not closed. For this purpose, the algorithm uses a loop for all of the possible suffixes of cex from line 2 to line 10. For each of the suffix e, the algorithm clones the observation table (S, E, T) and stores in OT (line 3). This is because processing with OT will not affect the current (S, E, T). After that, the algorithm adds e to E of OT, updates OT, and checks if OT is closed (from line 5 to line 7). If the updated OT is not closed, then adding e to E of (S, E, T) will make (S, E, T) not closed. In order to make it closed, an *sa* will need to be added to S. This will make the next conjecture C' different from the previous conjecture C. As a result, Algorithm 1 can continue learning assumption. Therefore, Algorithm 3 returns *yes* and e as the needed suffix for adding to the input observation table (line 8). If there is no e that can make the observation table not closed, then the algorithm returns *no*. This means that Algorithm 1 will come into an infinite loop. We will need to find another solution so that the algorithm can continue running correctly. This kind of solution will be one of our future work and not be mentioned in this paper.

3.2 Reducing the Number of Membership Queries

Although the method shown in Algorithm 3 can prevent Algorithm 1 from running infinitely in some special cases, it costs us more time to do that than Algorithm 1. We propose an algorithm to improve the performance of the whole learning process by reducing the number of membership queries. Although this improvement seems to be trivial and obvious when implementing the assume-guarantee reasoning, but with the large number of membership queries can be reduced and in the context of software evolution where the software needs to be rechecked whenever there is any changes, the improvement can play an important role in reducing the cost of software verification in practice. Details of the

Algorithm 4: Improved Observation Table Update

```
input : An observation table $(S, E, T)$, the Membership queries result
        dictionary $dict$
output: The updated observation table $(S, E, T)$
1  begin
2      forall the $s \in S$ or $sa \in S$ do
3          forall the $e \in E$ do
4              $str \leftarrow s.e$ or $str \leftarrow sa.e$
5              if $dict$ contains $str$ then
6                  $t \leftarrow$ get value of $str$ from $dict$
7                  Update the corresponding $T$ in $(S, E, T)$ with $t$
8              else
9                  $t \leftarrow$ ask membership query for $str$
10                 Store $\langle str, t\rangle$ to $dict$
11                 Update the corresponding $T$ in $(S, E, T)$ with $t$
12             end
13         end
14     end
15 end
```

algorithm is shown in Algorithm 4. For this purpose, we use a dictionary *dict* to store list of query results in form of couple $\langle str, t\rangle$, where *str* is the trace that is passed to *teacher* as a membership query and t is the corresponding result. For each *str* to be passed to *teacher* as a membership query, if it exists in *dict*, then its value will be used to update the observation table (line 5 to line 7) without asking a new membership query result to *teacher*. Otherwise, it will ask a new membership query to *teacher*, store the result to *dict*, and update the observation table (line 8 to line 11). *dict* will be used throughout the assumption learning process to improve the learning performance.

4 Experiments

We have implemented the two improvements in Algorithm 3 and Algorithm 4 into an application called IAGTool[1] in order to compare assumption generation performance of the original in [5] and improved algorithms. The algorithm is developed using Microsoft Visual Studio 2015 Community [10]. The test is carried 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 Home; IDE: Visual Studio Community 2015. The experimental results are shown in Table 1. In this table, there are three kinds of results of the original learning algorithm without Algorithm 3 (denoted by *Original*), original learning algorithm with Algorithm 3 (denoted by *Original+*), and improved learning algorithm with Algorithm 3 and 4 (denoted by *Improved*) to compare for each of test cases (with safety property p). The number of membership queries shown in column "Queries Number". That allow us to calculate how many queries are saved using the improved algorithm. The "Time (ms)" columns show us how much time the improved algorithm and the original one take to generate assumptions. "-" values indicate cases where the learning algorithm failed to generate required assumptions. Among the test cases shown in Table 1, "TestCase1" is the example presented in [5]. From the experimental results shown in Table 1, we have the following observations:

Table 1. Experimental results

No.	M1	M2	p	Original		Original+		Improved	
				Queries Number	Time (ms)	Queries Number	Time (ms)	Queries Number	Time (ms)
TestCase1	3	3	2	52	4	52	3	17	1
TestCase2	42	5	3	-	-	875	10278	161	4032
TestCase3	23	11	6	-	-	465	8889	82	3380
TestCase4	78	22	7	78	13495	78	7361	29	3981
TestCase5	30	12	3	-	-	2440	11064	339	3982
TestCase6	65	32	3	-	-	1421	651131	229	276253

– The *Original* algorithm can only generate assumptions for TestCase1 and TestCase4. In most of other test cases, the algorithm failed to generate assumption correctly due to the infinite loop during the learning process. In

[1] http://www.coltech.vnu.edu.vn/∼hungpn/IAGTool/

the meantime, when using the improved counterexample analysis algorithm, both of the *Original+* and *Improved* algorithms successfully pass the loop and are able to generate the required assumptions for these test cases.

– The number of membership queries is dramatically reduced by using Algorithm 4. There is no test case where the number of membership queries in *Original+* is the same or less than that of the *Improved* algorithms because even with the smallest test case (TestCase1), the learning process needs two closed observation tables during assumption generation. Therefore, several membership queries have been saved thanks to Algorithm 4.
– The *Improved* algorithm runs much faster than the *Original+* algorithm. This is because Algorithm 4 implemented in *Improved* algorithm has saved several membership queries during the learning process. As a result, generating assumptions of the *Improved* algorithm has faster speed than the original algorithm in [5].
– When running with such small test cases as shown in Table 1, the *Improved* algorithm runs much faster than the *Original+* one. Therefore, in practice, it could improve the assumption generation speed dramatically.
– In order to implement Algorithm 4, we need to create a dictionary to store membership query results. That costs us some more memory. However, with the current hardware technology, this will be a cost-effective method to improve the whole speed of verification process.

5 Related Works

There are a lot of researches related to optimizing the L^* based assume-guarantee verification. Consider only the most current works, we can refer to [2,3,6–8,11].

Chaki and Strichman proposed three optimizations in [2] to the L^* based automated Assume-Guarantee reasoning algorithm for the compositional verification of concurrent systems. The paper suggested an optimization that uses some informations that are already available to *teacher* in order to avoid many unnecessary membership and candidate queries. Sharing concern about improving the assumption generation speed, our researches proposed two improvements on this. The first one is to improve suffix choosing process that will prevent the original assumption generation method from coming into an infinite loop if choosing the last action *cex* as suffix. The second one is to reduce the number of membership queries by another kind of observation that traces submitted to *teacher* for membership queries are duplicate many times.

In a series of papers of [7,8,11], Hung et al. proposed a method for generating minimal assumptions, improving, and optimizing that method to generate those assumptions for compositional verification. However, that is for the result of the verification, not improve the method to generate the assumption itself. This paper shares the interest of improving the compositional verification, but we focus on improving the method itself so that it has faster speed than the original one.

In 2010, Chen et al. proposed a pure method for learning assumption through implicit learning in [3]. This has a great result on having faster speed than the original assumption generation method. Nevertheless, it focuses on a brand new approach that uses a specification method with Boolean functions. We share the interest about compositional verification, but we focus on the original assumption generation method to improve it in order to prevent it from running infinitely if choosing the last action *cex* as suffix and to have the faster speed.

In [6], 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 assumption for assume-guarantee reasoning for hardware verification. Our approach focuses on the original assumption generation method to improve it by reducing the number of membership queries and improving the counterexample analyzing algorithm to choose correct suffix that prevent the algorithm from running infinitely if choosing the last action *cex* as suffix.

In [9], Maler and Pnueli have mentioned the idea of analyzing the counterexample when learning infinitary Regular Sets. We share the idea of analyzing counterexample when implementing the L^* algorithm, but we apply it for the context of assume-guarantee reasoning. With this small finding, our proposed algorithm makes it easier for the implementation of software verification. Although it seems to be small change, but it can prevent the original algorithm from running infinitely if choosing the last action *cex* as suffix.

6 Conclusion

In order for the assume-guarantee paradigm to be used effectively in practice with large-scale systems, its assumption generation time must be improved as much as possible. We have presented a method to do this by preventing *learner* from asking membership queries for traces that are already been asked. This is done by creating a dictionary to store membership queries results and use it whenever *learner* wants to ask membership queries. *Learner* can only ask membership query for a trace that has never been asked. We have also applied the idea for choosing suffixes from counterexamples in [9] in the implementation of assume-guarantee reasoning so that the learning process will not run endlessly if we simply consider the last action of *cex* as suffix. This is done by trying to add suffixes with length from one to the length of *cex* to E of the observation table. If a suffix can make the observation not closed, that is the one to be added to E. The experimental results included in this paper also show that the proposed method have improved the assumption generation time significantly and there is no infinite loop when learning assumptions for the presented test cases.

Although the presented methods have a very positive effect on the assumption learning process, there are a lot of things need to be done. The first one is already described in Section 3.1. The solution in Algorithm 1 will also not be able to run correctly when Algorithm 3 returns *no*. We will need another research to analyze such cases to find out a solution so that we can generate the required assumption if it exists. The second one is that we are analyzing if there is any case where the learning process faces infinite loop. The third one is that the proposed framework in [5] is not for evolving systems. What needs to be done in order to generate the best assumption for evolving systems. Additionally, we are also in process of applying the proposed algorithms to larger systems in practice in order to verify their correctness and usefulness when doing verification.

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'07. Proceedings*, chapter Optimized L*-Based Assume-Guarantee Reasoning, pages 276–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
3. Y.-F. Chen, E. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, and B.-Y. Wang. Automated assume-guarantee reasoning through implicit learning. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 511–526. Springer Berlin Heidelberg, 2010.
4. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.
5. 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.
6. A. Gupta, K. L. Mcmillan, and Z. Fu. Automated assumption generation for compositional verification. *Form. Methods Syst. Des.*, 32(3):285–301, June 2008.
7. 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)*, pages 1–6, Feb 2012.
8. 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.
9. O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Information and Computation*, 118(2):316 – 326, 1995.
10. Microsoft. Visual studio community 2015. https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx.
11. P. Ngoc Hung, T. Aoki, and T. Katayama. *Theoretical Aspects of Computing - ICTAC'09: 6th International Colloquium. Proceedings*, chapter A Minimized Assumption Generation Method for Component-Based Software Verification, pages 277–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
12. 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.