On Implementation of the Improved Assume-Guarantee Verification Method for Timed Systems[*]

Hoang-Viet Tran
15028003@vnu.edu.vn
Faculty of Information Technology
VNU University of Engineering and Technology
Hanoi, Vietnam

Quang-Trung Nguyen
trungnq@vcu.edu.vn
Vietnam University of Commerce
Hanoi, Vietnam

Pham Ngoc Hung
hungpn@vnu.edu.vn
Faculty of Information Technology
VNU University of Engineering and Technology
Hanoi, Vietnam

ABSTRACT

The two-phase assume-guarantee verification method for timed systems using TL^* algorithm implemented in the *learner* has been known as a potential method to solve the problem of *state space explosion* in model checking thanks to its *divide and conquer* strategy. This paper presents three improvements to the verification method. First, we remove the untimed verification phase from the verification process. This removal reduces the time complexity of the verification process because of the great time complexity of this phase. Second, we introduce a *maxbound* to the equivalence queries answering algorithm implemented in the *teacher* which acts as a method for the *teacher* to return "*don't know*" results to the *learner* to prevent the verification process from many endless scenarios. Finally, we introduce a technique to analyze the counterexample received from the *teacher* and another one implemented in the equivalence queries answering algorithm which helps the *teacher* not return a counterexample that has been returned to the *learner*. This technique keeps the verification process from running forever in several circumstances. We give primitive experimental results for both two-phase assumption generation method and the improved one with some discussions in the paper.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**.

KEYWORDS

Software verfication, timed systems, assume-guarantee verification, component-based software

ACM Reference Format:
Hoang-Viet Tran, Quang-Trung Nguyen, and Pham Ngoc Hung. 2019. On Implementation of the Improved Assume-Guarantee Verification Method for Timed Systems. In *Soict '19: The Tenth International Symposium on Information and Communication Technology, December 4 – 6, 2019, Hanoi - Ha Long Bay, Vietnam.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3368926.3369659

[*]This paper is an extension of the paper [25] presented at KSE'10.

1 INTRODUCTION

In modern software development, quality assurance in general and correctness verification in particular play an important role, especially for the timed systems correctness verification since these systems have much higher complexity than that of untimed ones. As a result, reducing time complexity of existing verification methods for timed systems emerges as a big challenge for software industry. Among verification methods, model checking [4, 5, 23] has gained a lot of attention as the most promising approach thanks to its fully automatic basis. However, one well known issue of model checking is the *state space explosion* problem [4, 23] when checking large-scale systems. Assume-guarantee reasoning (AGR) [3, 12, 22] is a method developed to solve the *state space explosion* problem of model checking. Despite its potential application in verifying large-scale software, the method has not been used much in software industry due to its high time complexity. This fact becomes clearer when doing verification of not only untimed systems but also timed ones due to the truth that the state space of timed systems is much bigger than that of untimed ones.

To our knowledge, Lin et al. is the first one who proposes a method to apply AGR for timed systems in a fully automatic manner [19]. The method contains two phases of learning assumption in which the first one is to generate untimed assumption while the second one generates the required timed assumption. If an untimed assumption can be generated in the first phase, it will be used as the input of the second phase to generate the required timed assumption. However, this method (hereafter called the two-phase assumption generation method) has some limitations as follows. The first one is that both phases of the learning process have high time complexity. This makes the method have great time complexity. The second one is that dividing the learning process into two phases does not help the method cover all the cases where an assumption can be generated. This problem exists because of the fact that there are two types of assume-guarantee reasoning rule which are used in software verification: circular assume-guarantee rule (CIRC-AG) which is sound and complete; non-circular assume-guarantee rule (NC-AG) which is not complete [14]. When applying the NC-AG rule, most of the proposed algorithms return either $YES{+}assumption$ or $NO{+}counterexample$ [2, 6, 11, 19]. Apparently, these algorithms implicitly assume that the required assumption exists and the *teacher* knows the assumption when solving membership queries and equivalence queries from the *learner*. However, in general, there exists an area where the *teacher* does not know if the assumption exists. In a typical scenario where a given system M satisfies a predefined safety property p, but the *teacher* does not know if the assumption which satisfies NC-AG rule exists. As a result, the learning process introduced in [2, 6, 11, 19] can run endlessly. Consequently, we need to add a guide for the *teacher* to return *"don't know"* result so that the *learner* can stop the learning process. We realize this idea by adding a kind of bound called *maxbound* to the equivalence queries answering algorithm implemented in the *teacher*. The learning process can be presented as shown in Figure 1. Consider one of the "*don't know*" scenarios where the two-phase

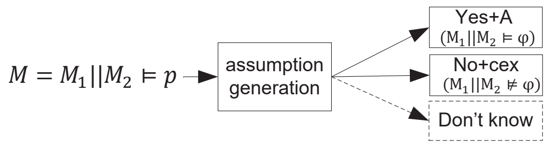

Figure 1: The general NC-AG verification.

assumption generation method proposed by Lin et al. [19] does not cover. Given a timed system M with two components M_1 and M_2 (i.e., $M = M_1 \parallel M_2$) and a safety property p. Assume that the assumption which satisfies the NC-AG rule exists but the *teacher* does not know about this. Therefore, we have a case where the untimed assumption generated by untimed learning phase leads the timed learning phase to the conclusion of "*don't know*". In this scenario, the untimed learning phase appears to be redundant in spite of its high time complexity. On the other hand, there exist chances where we can prevent the learning process from wrong direction by starting the learning process from the beginning (i.e., from λ). Consequently, a number of "*don't know*" scenarios from the two-phase assumption generation method can be covered. In addition, the removal of the untimed learning phase also improves the speed of the learning process thanks to its high time complexity.

From the above analysis, this paper presents three improvements to the two-phase assumption generation method. The key idea of the first improvement is to remove the untimed learning phase from the learning process to get an improved algorithm which contains only the timed learning phase (hereafter called one-phase assumption generation method). This removal reduces time complexity of the verification process and makes the process cover some "*don't know*" cases where the two-phase assumption generation method does not. The second improvement adds a *maxbound* to the equivalence queries answering algorithm which acts as a guide for the *teacher* to return "*don't know*" result to the *learner*. The *learner*, in its turn, will stop the learning process when receiving a "*don't know*" result from the *teacher*. The last improvement is a technique to analyze the counterexample received from the *teacher* for finding a better assumption candidate for the next learning iteration. This improvement also includes a technique implemented in the equivalence queries answering algorithm which helps the *teacher* not return a counterexample that has already been returned to the *learner*. The two techniques implemented in both the *learner* and the *teacher* prevent the learning process from several endless scenarios. Another result of this improvement is that the *teacher* can find and return a new "*don't know*" case where it does not know if the assumption exists or not. This is the case where no more counterexample can be found to return to the *leaner* for generating a better assumption candidate. Our initial experimental results show that the one-phase assumption generation method outperforms the two-phase one in term of time and in most of the test scenarios. We give discussions about the experiment results in the paper.

The rest of the paper is structured as follows. Section 2 shows our motivations for doing the research. Section 3 shows the one-phase assumption generation method which contains the technique to analyze the counterexample returned from the *teacher*. A variant of the equivalence queries answering algorithm is shown in Section 4 which contains the idea not to return a counterexample which has already returned to the *learner*. We give experimental results and discussions in Section 5. Related researches to this paper is shown in Section 6. The paper is concluded in Section 7.

2 MOTIVATION

The two-phase assumption generation method was introduced by Lin et al. [19] in 2014. This section gives discussions about *"don't know"* coverage of the two-phase assumption generation method. Although the two-phase assumption generation method can nicely generate the required assumption in general, it does not cover all the cases. There are many scenarios where $M \models p$ but the *teacher* does not know if returning a new counterexample can help the learning process to generate the needed assumption. This can result in an endless learning process as the one shown in Section 3.1. Consequently, we need a kind of *maxbound* to help the *teacher* to return *"don't know"* result so that the *learner* can stop the learning process. This *maxbound* is integrated into the equivalence queries answering algorithm implemented in the *teacher* as shown in Algorithm 2.

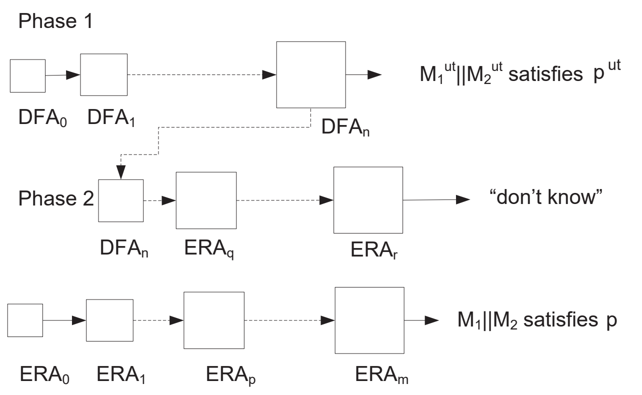

Figure 2: $M_1^{ut} \parallel M_2^{ut} \models p^{ut}$, but phase 2 result is *"don't know"*.

Consider an example shown in Figure 2 where a given system $M = M_1 \parallel M_2$ satisfies a predefined safety property p (i.e., $M \models p$). Although $M \models p$ and phase 1 successfully generate untimed assumption at step n (i.e., $M_1^{ut} \parallel M_2^{ut} \models p^{ut}$), phase 2 ends up with a *"don't know"* situation. However, if we try to learn from the beginning without having to learn untimed assumption (i.e., learning from an empty string (λ) or from ERA_0), the result may be different (i.e., "$M \models p$" is returned). This is because we have some other chances to follow a correct direction during the learning process. In this scenario, the result of untimed learning phase can lead the learning process to a direction where the *teacher* does not know what counterexample for the *learner* to generate a better assumption candidate in the next learning iteration. Moreover, this learning phase has a high time complexity. Therefore, in our opinion, this phase should be removed from the learning process.

From this observation, we propose an algorithm that learns an assumption from the beginning (from λ) in only one phase by removing the untimed learning phase. This not only saves the running time of the untimed phase, but also lets the learning process have some other chances of going to a correct learning direction. This makes the proposed learning algorithm cover some of the *"don't know"* scenarios where the two-phase learning algorithm does not.

3 ONE-PHASE ASSUMPTION GENERATION

For more information about the background concepts, please refer to Lin et al.'s paper [19]. Hereafter, we use the following NC-AG rule in which M, M_1, M_2, and p are represented by event-recording automata (ERA).

Definition 3.1. (The NC-AG Rule). Given a timed system $M = M_1 \parallel M_2$ and a predefined property p, if M_1 satisfies p under an assumption A and M_2 guarantees A, then $M \models p$.

$$\frac{\begin{array}{r} M_1 \parallel A \models p \\ M_2 \models A \end{array}}{M_1 \parallel M_2 \models p}$$

When implementing the one-phase assumption generation method proposed in the paper [25], we saw that if we simply implement the equivalence queries answering algorithm proposed in *teacher*, there are several scenarios where the learning process can run endlessly. Furthermore, both the two-phase and one-phase assumption generation methods [19, 25] did not describe in details how to get a suffix t from a given counterexample *cex* to generate a better assumption candidate. This section shows an example of an endless learning process and a variant of the one-phase assumption learning algorithm that contains a technique to find the suffix t.

3.1 Example for an Endless Learning Process

Consider a motivation example where the verification goes endlessly with the current algorithms [19, 25] implemented in the *learner* and *teacher*. This is why we need some improvements in both sides so that the learning process can go further when doing verification. Consider a flexible manufacturing system (FSM) [7] which consists of five components: one conveyor, one mill, two robots, and one assembly station. The property requires that the system must have output after the input within three units of time. We divided the system into two components of M_1 and M_2 using a heuristic where only components containing behaviors included in the property are considered. The result system $M = M_1 \parallel M_2$ and its property p are shown in Figure 3.

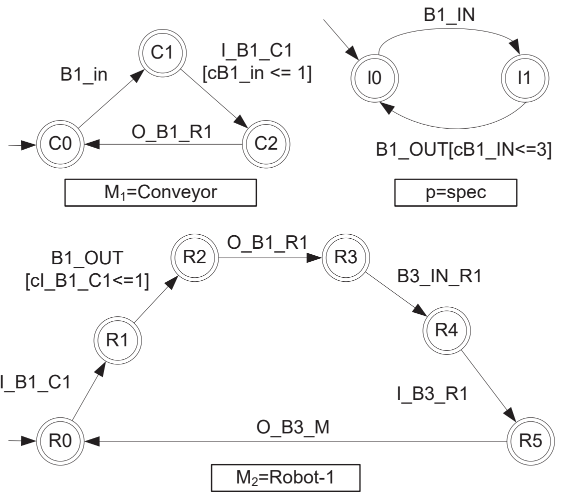

Figure 3: An example for an endless learning process.

Table 1: One observation table in verification process of M

		(λ,true)
S	(λ,true)	1
S	(B1_OUT,true)	0
SA	(I_B1_C1,cB1_IN>1)	1
SA	(I_B1_C1,cB1_IN<=1)	1
SA	(O_B1_R1,true)	1
SA	(B1_OUT,true)	0

When learning, the observation table as shown in Table 1 is created. The corresponding assumption is shown in Figure 4. However, when asking the *teacher* an equivalence query with that assumption candidate, the *teacher* returns a counterexample *cex=(I_B1_C1,cB1_IN<=1)(O_B1_R1,true)(I_B1_C1,cB1_IN<=1)*. After analyzing *cex*, no suffix t can help the *learner* to generate a better candidate. As a result, the same candidate as shown in Figure 4 is

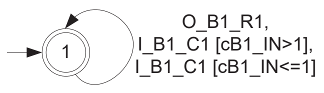

Figure 4: The corresponding assumption candidate.

used to ask a new equivalence query to the *teacher*. With no change in the equivalence queries answering algorithm implemented in the *teacher* as proposed by Lin et al. [19], the same counterexample *cex* will be returned. Consequently, the learning process will run forever. If the equivalence queries answering algorithm proposed by us [25] is implemented, the learning process can be stopped when *maxbound* is reached. Nevertheless, that is not the best we can do to implement the verification method. In sections below, we present variants of both one-phase assumption learning and equivalence queries answering algorithms that make the learning process go further to reach a decisive result.

3.2 A Variant of the One-Phase Assumption Learning Algorithm

As mentioned in Section 3.1, both the two-phase and one-phase learning algorithms do not give detailed steps of finding a suffix t to generate a better assumption candidate. We present a variant of the one-phase learning algorithm that includes a technique to find t. This technique shares the same idea of the one proposed by Le et al. [15]. However, the idea is now applied to the one-phase assumption generation method. Details of the variant algorithm is shown in Algorithm 1. The algorithm starts by initializing the observation table (S, E, T) with $S = E = \{(\lambda, true)\}$ (line 2). The learning process goes to the main iteration from line 3 to 39. Then, it updates (S, E, T) by using timed membership queries (line 4). While (S, E, T) is not closed, the algorithm tries to make (S, E, T) closed by using timed membership queries (lines 5 to 8). When (S, E, T) is closed, the algorithm constructs an assumption candidate C (line 9), asks the *teacher* an equivalence query, and stores the result in *EQResult* (line 10). If *EQResult* is *yes*, the algorithm returns C as the needed assumption and reports "$M = M_1 \parallel M_2 \models p$" (line 12). In case *EQResult* is $(continue, cex)$, the algorithm needs to update (S, E, T) using *cex* in order to have a better assumption candidate (from line 13 to 33). Let $cex = (a_1, g_1)(a_2, g_2)...(a_n, g_n)$ (line 14). With each timed action (a_i, g_i) in *cex*, if there exists another timed action (a_i, g) that is a substring of $s \in S \cup S.\Sigma_T$ or $e \in E$ and $[\![g_i]\!] \subset [\![g]\!]$ (line 16), then we will split the timed action (a_i, g) as follows. Let $G = \{\hat{g_1}, \hat{g_2}, ..., \hat{g_m}\}$ be the set of constraints that contains the result of the constraints subtraction of $[\![g]\!] - [\![g_i]\!]$ (line 17). The timed actions set of Σ_T is then updated by replacing $\{(a_i, g)\}$ by $\{(a_i, g_i), (a_i, \hat{g_1}), (a_i, \hat{g_2}), ..., (a_i, \hat{g_m})\}$ (line 18). Then, with $s \in S \cup S.\Sigma_T$, s is split into $\{\hat{s_0}, \hat{s_1}, \hat{s_2}, ..., \hat{s_m}\}$ where (a_i, g_i) is a substring of $\hat{s_0}$, $(a_i, \hat{g_j})$ is a substring of $\hat{s_j}$, $\forall j \in \{1, 2, ..., m\}$ (line 19). With $e \in E$, e is split into $\{\hat{e_0}, \hat{e_1}, \hat{e_2}, ..., \hat{e_m}\}$ where (a_i, g_i) is a substring of $\hat{e_0}$, $(e_i, \hat{g_j})$ is a substring of $\hat{e_j}$, $\forall j \in \{1, 2, ..., m\}$ (line 20). (S, E, T) is then updated for the newly added cells (line 21). Let *count* be the number of timed actions in *cex* (line 24). To find a suitable suffix t, the algorithm considers every suffix t of *cex* that has the number of timed actions k from 1 to *count* (line 25 to 33). With each t, the algorithm tries, with a copy *OT* of (S, E, T) (line 26), adding t to *OT*'s E and updating *OT* (line 28). If the updated *OT* is closed, meaning that adding t to E does not result in a better assumption candidate, we need to consider another suffix. Otherwise, the updated *OT* is not closed, meaning that adding t to E results in a better assumption candidate for the learning process, the loop of finding t is stopped and t will be added to the suffixes set E of (S, E, T) (line 29 to 32). After adding t to E (line 30), the current loop of finding t is stopped (line 31) and the learning process comes back to step 4 to find a better assumption candidate. In case *EQResult* is *"don't know"*, the algorithm stops and reports *"don't know"* (line 35). The last case is when *EQResult* is (no, cex), the algorithm stops and report "$M = M_1 \parallel M_2 \not\models p$" + cex (line 37).

In regards to the correctness of Algorithm 1, please refer to our previous paper [25]. In this section, we only discuss about the steps of finding t from line 24 to 33. Although these steps cannot always find out the suitable suffix t to be added to E, they give a feasible method to the implementation of the learning process. Moreover, when t cannot be found from these steps, because Algorithm 1 will be used together with the *teacher* which is implemented using a variant of the equivalence queries answering algorithm presented in Section 4, the learning process can go further in comparison with the learning process that uses either the one-phase [25] or the two-phase learning algorithm [19].

4 EQUIVALENCE QUERIES ANALYSIS

For the technique presented in Section 3.2 to be used effectively, we need a variant of the equivalence queries answering algorithm which does not return counterexample that has already been returned to the *learner*. The equivalence queries answering algorithm was originally proposed by Lin et al. [19] and later improved by us [25]. The variant algorithm is shown in Algorithm 2. For the purpose of managing counterexamples that have already been returned to the *learner*, the algorithm maintains a list called *ReturnedCexList* which contains those counterexamples. When the algorithm has found a counterexample *cex* to be returned to the *learner*, it checks if *cex* is in *ReturnedCexList*. If yes, the algorithm continues finding if there exists another *cex* to return to the *learner*. Otherwise, *cex* is returned to the *learner* for learning a better assumption candidate. The algorithm accepts an assumption candidate C as input and returns either *yes* (i.e., C satisfies NC-AG rule) or $(continue, cex)$ (i.e., C does not satisfy NC-AG rule, but the *learner* can use the returned *cex* to learn a better assumption candidate) or (no, cex) (i.e., C does not satisfy NC-AG rule but the *learner* cannot learn another better candidate from *cex* because $M_1 \parallel M_2 \not\models p$) or "*don't know*" (i.e., C does not satisfy NC-AG rule and the *teacher* does not know if continuing the learning process can reach a decisive result or not). The algorithm starts by checking if $L(M_1 \parallel C \parallel \overline{p}) = \emptyset$ (line 2). If yes (i.e., $M_1 \parallel C \models p$), the algorithm continues checking if $L(M_2 \parallel \overline{C}) = \emptyset$ (line 3). If yes (i.e., $M_2 \models C$), the algorithm returns *yes* to the *learner*. If $M_2 \not\models C$, let *cex* be one trace in $L(M_2 \parallel \overline{C})$ (line 6) and its projected word cex' over Σ has never been returned to the *learner* (i.e., $cex' = cex_{\downarrow_\Sigma} \notin ReturnedCexList$) (line 6). If such *cex* does not exist (line 7), meaning that all traces in $L(M_2 \parallel \overline{C})$ have already been returned to the *learner* but no better assumption candidate can be found. The *teacher* does not know how to find another needed counterexample. The *teacher* returns "*don't know*" to stop the learning process (line 8). If such *cex* exists, the algorithm adds cex' to *ReturnedCexList* (line 10). Then, the algorithm analyzes if *cex* belongs to $L(M_1 \parallel \overline{p})$ (line 11). If yes (i.e., $M_1 || M_2 \not\models p$), the algorithm returns (no, cex') (line 12). In case $cex \notin L(M_1 \parallel \overline{p})$, the algorithm increases the number of equivalence queries processed *QcNum* by 1 (line 14) and checks if *QcNum* reaches *maxbound* (line 15). If yes, the algorithm returns "*don't know*" to the *learner* (line 16). Otherwise, the algorithm gets $cex' = cex_{\downarrow_\Sigma}$ and returns the result $(continue, cex')$ to the *leaner* (line 18 to 19). In case $L(M_1 \parallel C \parallel \overline{p} \neq \emptyset)$, let *cex* be one trace in $L(M_1 \parallel C \parallel \overline{p})$ and $cex' = cex_{\downarrow_\Sigma} \notin ReturnedCexList$ (line 23). The same as the above case, if such *cex* does not exist, the algorithm returns "*don't know*" to the *learner* to stop the learning process (line 25). If such *cex* exists, the algorithm adds $cex' = cex_{\downarrow_\Sigma}$ to *ReturnedCexList* (line 27). Then, the algorithm checks if $cex \in L(M_2)$. If yes (i.e., $M_1 \parallel M_2 \not\models p$), the algorithm returns (no, cex') to *learner* (line 29). In case $cex \notin L(M_2)$, the algorithm increases *QcNum* by 1 (line 31) and checks if *QcNum* reaches *maxbound*. If *maxbound* is reached (line 32), the *teacher*

Algorithm 1: One-Phase assumption Learning Algorithm

```
1  begin
2      Initialize $(S, E, T)$ with $S = E = \{(\lambda, true)\}$.
3      while $true$ do
4          Update $(S, E, T)$ using $Q_m$ queries.
5          while $\exists(s.a)|\ row(s.a) \not\equiv row(s'), \forall s' \in S, a \in \Sigma_T$ do
6              $S \leftarrow S \cup \{(s.a)\}$.
7              Update $(S, E, T)$ by using $Q_m((s.a).b.e)\ \forall b \in \Sigma_T$ and $e \in E$.
8          end
9          Construct an ERA assumption candidate $C$ from $(S, E, T)$.
10         $EQResult \leftarrow$ Ask an equivalence query for $C$
11         if $EQResult = yes$ then
12             return $C$ as needed assumption and report "$M = M_1 \parallel M_2 \models p$".
13         else if $EQResult = continue$ then
14             Let $cex = (a_1, g_1)(a_2, g_2)...(a_n, g_n) \leftarrow$ the counterexample from $Teacher$.
15             foreach $(a_i, g_i), i \in \{1, 2, ...n\}$ do
16                 if $(a_i, g)$ is a substring of $s \in S \cup (S.\Sigma_T)$ or $e \in E$ such that $[\![g_i]\!] \subset [\![g]\!]$ then
17                     $G = \{\hat{g_1}, \hat{g_2}, ..., \hat{g_m}\} \leftarrow [\![g]\!] - [\![g_i]\!]$.
18                     $\Sigma_T = \Sigma_T \backslash \{(a_i, g)\} \cup \{(a_i, g_i), (a_i, \hat{g_1}), (a_i, \hat{g_2}), ..., (a_i, \hat{g_m})\}$.
19                     Split $s$ into $\{\hat{s_0}, \hat{s_1}, \hat{s_2}, ..., \hat{s_m}\}$, where $(a_i, g_i)$ is a substring of $\hat{s_0}$, $(a_i, \hat{g_j})$ is a substring of $\hat{s_j}$, $\forall j \in \{1, 2, ..., m\}$.
20                     Split $e$ into $\{\hat{e_0}, \hat{e_1}, \hat{e_2}, ..., \hat{e_m}\}$, where $(a_i, g_i)$ is a substring of $\hat{e_0}$, $(e_i, \hat{g_j})$ is a substring of $\hat{e_j}$, $\forall j \in \{1, 2, ..., m\}$.
21                     Update $(S, E, T)$ for newly added cells using $Q_m$.
22                 end
23             end
24             $count \leftarrow$ number of timed actions in $cex$.
25             for $k = 1$ to $count$ do
26                 $OT \leftarrow$ A copy of $(S, E, T)$.
27                 $t \leftarrow$ A suffix of $k$ timed actions in $cex$.
28                 Add $t$ to $OT$'s $E$; Update $OT$ using timed membership queries.
29                 if $OT$ is not closed then
30                     Add $t$ to $(S, E, T)$'s $E$.
31                     break.
32                 end
33             end
34         else if $EQResult$ = "$don't\ know$" then
35             return "$don't\ know$" and stop.
36         else
37             return "$M = M_1 \parallel M_2 \not\models p$" + $cex$ and stop. // $Teacher\ returns\ (no + cex)$
38         end
39     end
40 end
```

returns "*don't know*" to the *learner* to stop the learning process (line 33). Otherwise, the algorithm gets $cex' = cex_{\downarrow_\Sigma}$ (line 35) and returns $(continue, cex')$ to the *learner* (line 36).

Correctness discussion The correctness of the original proposed algorithm has been proved in the paper [25]. In the limited scope of this paper, we give discussions about the main differences between Algorithm 1, 2 and those proposed by us [25]. The differences reside in both Algorithm 1 and 2. In Algorithm 1, the difference is included in steps from line 24 to line 33. Although these are simple steps, the algorithm gives a technique in details to the implementation of finding a suitable suffix t to be added to the suffixes set E. The suffix t must be the one that when being added to (S, E, T), a better assumption candidate can be generated for the next learning iteration. If such t cannot be found, the learning algorithm relies on the *teacher* to have another counterexample cex to find a suitable t. When *teacher* processes equivalence queries in Algorithm 2, in its turn, in case the same candidate is received, if a certain counterexample cex' which has already been returned to the *learner* (i.e., $cex' \in ReturnedCexList$), the algorithm tries to find another one (lines 6 to 10 or lines 23 to line 27). With this method of implementation, the combination of Algorithm 1 and 2 makes the learning process go further toward a decisive result. Another result

Algorithm 2: A variant of the equivalence queries answering algorithm $Q_c(C)$

```
Input: $C$: the ERA assumption candidate; $maxbound$: the maximum number of candidate queries $Teacher$ will answer
Output: $yes/(continue, cex)/(no, cex)/don't\ know$
1  begin
2      if $L(M_1 \parallel C \parallel \overline{p} = \emptyset)$ then
3          if $L(M_2 \parallel \overline{C}) = \emptyset$ then
4              return $yes$ .
5          else
6              Let $cex \in L(M_2 \parallel \overline{C})$ where
                 $cex' = cex_{\downarrow_\Sigma} \notin ReturnedCexList$.
7              if $cex$ does not exist then
8                  return "don't know".
9              end
10             $ReturnedCexList \leftarrow ReturnedCexList \cup \{cex'\}$.
11             if $cex \in L(M_1 \parallel \overline{p})$ then
12                 return $(no, cex')$.
13             else
14                 $QcNum \leftarrow QcNum + 1$.
15                 if $QcNum = maxbound$ then
16                     return "don't know".
17                 end
18                 $cex' \leftarrow cex_{\downarrow_\Sigma}$ and $cex' = (a_1, g_1)...(a_m, g_m)$.
19                 return $(continue, cex')$.
20             end
21         end
22     else
23         Let $cex \in L(M_1 \parallel \overline{p} \parallel C)$ where
             $cex' = cex_{\downarrow_\Sigma} \notin ReturnedCexList$.
24         if $cex$ does not exist then
25             return "don't know".
26         end
27         $ReturnedCexList \leftarrow ReturnedCexList \cup \{cex'\}$.
28         if $cex \in L(M_2)$ then
29             return $(no, cex')$.
30         else
31             $QcNum \leftarrow QcNum + 1$.
32             if $QcNum = maxbound$ then
33                 return "don't know".
34             end
35             $cex' \leftarrow cex_{\downarrow_\Sigma}$ and $cex' = (a_1, g_1)...(a_m, g_m)$.
36             return $(continue, cex')$
37         end
38     end
39 end
```

of this combination is that the "*don't know*" situation appears to be clearer to the implementation. Now, we have two scenarios where "*don't know*" result is returned to the *learner*. First, the *maxbound* is reached when processing equivalence queries (line 16 or 33). Second, the *teacher* does not know what counterexample should be returned to the *learner* for learning a better assumption candidate (line 8 or 25). A minor different point in Algorithm 2 in comparison with the one proposed in our previous paper [25] is that we returned cex' right after projecting cex over Σ (line 18 to 36). This keeps the implementation simple but maintains its correctness as cex' is still the required correct counterexample.

5 EXPERIMENTS

We have implemented both two-phase [19] and one-phase (i.e., Algorithm 1) assumption generation methods in a tool called Timed systems verification tool (Tivet) to have assessments for both methods. Algorithm 2 is used as the equivalence queries answering algorithm for both cases. We have tested the tool with some common test data used in the research community listed below.

- **Client-server.** A client-server system [21] contains three components: one server and two clients. We have modified the system written by Magee and Kramer by adding timed constraints that each client must be released within two units of time being served and that it must send request after being released within two units of time. In the meantime, server must grant access to a client after receiving its request within one unit of time. The property requires that two clients must be served in order.
- **FMS.** A flexible manufacturing system (FMS) [7] is a system which produces blocks in which one block contains a cylindrical painted pin from raw blocks and pegs. We tested only the simplest version of *FMS* called *FMS-1* which contains one conveyor, one mill, two robots, and one assembly station. We tested this *FMS-1* with three properties called *spec1*, *spec2*, and *spec3*.
- **GSS.** A gas station system [13] is a combination of five components: one operator, one queue, one pump, and two customers. The station accepts these two customers for filling gas. Tested properties require that customers must be served in order and each of them must be able to start filling gas within three units of time after his payment. Those properties are modeled as *spec1*, *spec2*, and *spec3*.
- **Master-slave.** A master-slave system [21] contains two components: one master and one slave. We have modified the models so that the slave must be synchronized by the server within one unit of time after it starts and the master must be back to its own work after synchronizing the slave within two units of time. The property requires that the server must be back to its own work within three units of time after the slave starts.
- **Simple communication channel (ComChannel).** A simple communication channel which is proposed by Cobleigh et al. [6] and later modified by Lin et al. [19] contains two components: one input and one output. The property requires that the system must receive a new input after its output within five units of time and the output must be after receiving a new message within five units of time, too.

The tool including test data is available on http://www.tranhoangviet.name.vn/p/tivet.html. For more reliable results, we have run each test cases ten times and gotten the average time and memory used during verification process. The *maxbound* is chosen randomly for all test cases. Test results are shown in Table 2. In this table, columns denoted by "Systems", "$|C|$", "$|M_1|$", "$|M_2|$", "$|p|$", and "$|B|$" show the names of test systems, the size of clocks set, the size of M_1 (i.e., the number of locations in M_1), the size of M_2, the size of p, and *maxbound* used in the corresponding test cases, respectively. In Table 2, we can see there are some value of "$-$" in "$|M_1|$", "$|M_2|$", and "$|p|$" columns. These are the cases where test systems contains more than two components (i.e., $M = M_1 \parallel M_2 \parallel \ldots \parallel M_n$). In these cases, we implemented a kind of heuristic where M is divided into two components of H_1 and H_2 containing only behaviors included in the given property p (i.e., $M = H_1 \parallel H_2$). This method is based on an observation that those components which contain no behavior specified in the given property do not play any role in the system in regards to the property under checking. After applying the heuristic, the system is passed to either the two-phase or one-phase assumption generation method for processing. For each verification methods, we focus on the following main criteria to compare their efficiency: whether the given system satisfies its corresponding property; how many membership queries and equivalence queries it takes to reach a decisive result; how much time and memory it costs to reach the result. The test results retrieved when applying the two-phase assumption generation method are shown in the following columns: "Sat_2?", "$|A_2|$", "MQ_2^{ut}", "MQ_2^{t}", "EQ_2^{ut}", "EQ_2^{t}", "$Time_2$(ms)", and "Mem_2(B)". Those columns show the result if the given system satisfies its corresponding property, the size of the generated assumption, the number of untimed membership queries, the number of timed membership queries, the number of untimed equivalence queries, the number of timed equivalence queries, the time, and memory it takes to reach the decisive result, respectively. In these columns, the value "*DN*" means that the *teacher* has returned "*don't know*" when learning. In the meantime, test results retrieved when applying one-phase assumption generation method are shown in the following columns: "Sat_1?", "$|A_1|$", "MQ_1^{t}", "EQ_1^{t}", "$Time_1$(ms)", and "Mem_1(B). These columns show the result if the given system satisfies its corresponding property, the size of the generated assumption, the number of timed membership queries, the number of timed equivalence queries, the time, and memory it takes to reach the result, respectively. From the test results shown in Table 2, we have following discussions.

- In most of the cases (8 out of 9), one-phase assumption generation method takes less time to reach a decisive result than that of the two-phase assumption generation method. This is because in one-phase assumption generation method, we have removed the untimed learning phase which has high time complexity from the learning process.
- In regards to memory usage, there is no obvious difference between the two assumption generation methods. From the theory perspective, there is also no clear proof for that kind of difference.
- There is one case (*GSS_spec2*), the two-phase assumption generation method cannot reach a decisive result (i.e., meaning that the learner cannot say either $M \not\models p$ or $M \models p$) before reaching *"don't know" maxbound.* In the meantime, one-phase assumption generation method can reach the result of $M \not\models p$ in only 7 equivalence queries.
- There is one case (*MasterSlave*) where the time it takes to generate assumption using one-phase assumption generation method is longer than that when using the two-phase method. This is the case of a small system where the number of timed equivalence queries in one-phase method is 3 while that number in the two-phase method is 2. From this number, we can see another fact that timed equivalence queries take a lot of time to be processed.

6 RELATED WORKS

There are several researches which are related to the method of learning assumption for compositional verification of timed systems proposed by Lin et al. [19]. Some of those are about a tool called PAT that supports the method [8, 9]. Some other researches are about several aspects related to the Assume-Guarantee Reasoning verification method for software [1, 10, 16–18, 20, 24]. Our previous paper [25] in combination (hereafter called our paper) with this paper introduces three improvements to the two-phase verification method proposed by Lin et al. [19].

The two papers of Dong et al. [8, 9] present a tool called PAT that supports many methods in software analysis including the two-phase assumption generation method [19]. Although we shared the same interest about verification of timed systems, we give three improvements to the two-phase assumption generation method.

André et al. proposes a method to improve the verification of reachability properties in the full parametric systems [1]. Researches proposed by Li et al. [16, 17] gives methods for learning Büchi Automata and its application. In the meantime, other researches proposed by several authors [20, 24] present methods for synthesizing untimed and timed models for systems to be used in the verification process. Lin et al. introduces a method for tuning M_1, M_2, and partitioning the systems under checking [18]. The underlying ideas of compositional reasoning, foundational algorithms, and applications are summarized by Giannakopoulou et al. [10]. We share the same interest in applying the NC-AG verification to practice. However, we focus on improving the learning process itself and in the context of verification for timed systems.

7 CONCLUSION

The paper has presented three improvements to the two-phase assumption generation method proposed by Lin et al. [19]. The first improvement is to remove the untimed learning phase from the verification process which effectively reduces its time complexity. The second one is to give a *"don't know" maxbound* which helps the *teacher* to return "*don't know*" to the *learner* in cases the *teacher* does not know if returning the next counterexample can help the *learner* to generate a better candidate. This "*don't know*" result acts as a flag to stop the learning process. Last but not least, the paper introduces a technique to analyze the returned counterexample from the *teacher* to find a suitable suffix for learning a better assumption candidate. The paper also suggests another technique which helps the *teacher* not return a counterexample which has already been returned to the *leaner.* This improvement prevents the verification process from running endlessly in many scenarios.

Table 2: Experimental results

No.	Systems	$\lvert C\rvert$	$\lvert M_1\rvert$	$\lvert M_2\rvert$	$\lvert p\rvert$	$\lvert B\rvert$	Two-phase learning algorithm								One-phase learning algorithm					
							Sat_2?	$\lvert A_2\rvert$	MQ_2^{ut}	MQ_2^t	EQ_2^{ut}	EQ_2^t	$Time_2$(ms)	Mem_2(B)	Sat_1?	$\lvert A_1\rvert$	MQ_1^t	EQ_1^t	$Time_1$(ms)	Mem_1(B)
1	ClientServer	10	-	-	-	10	No	-	9	9	1	1	2,440	41,761,874	No	-	5	1	2,213	51,551,356
2	FMS-1_spec1	25	-	-	-	10	No	-	17	397	2	9	68,506	41,115,457	No	-	6	4	1,207	40,706,231
3	FMS-1_spec2	25	-	-	-	10	No	-	5	5	1	1	3,828	41,366,172	No	-	3	1	3,350	45,703,875
4	FMS-1_spec3	25	-	-	-	10	No	-	7	22	4	3	3,046	48,049,674	No	-	4	4	1,853	49,914,883
5	GSS_spec1	15	-	-	-	10	No	-	43	56	25	2	82,573	61,106,959	No	-	5	7	48,677	83,395,860
6	GSS_spec2	15	-	-	-	10	DN	-	43	323	25	10	128,026	19,373,386	No	-	5	7	39,282	18,383,930
7	GSS_spec3	15	-	-	-	10	Yes	2	5	20	4	5	5,279	9,600,567	Yes	1	3	5	3,778	11,666,328
8	MasterSlave	5	4	3	2	10	Yes	2	3	8	1	2	412	3,175,247	Yes	1	5	3	519	2,795,131
9	ComChannel	4	3	3	2	10	Yes	2	17	124	2	3	5,272	5,029,697	Yes	1	8	3	768	5,210,304

The initial experimental results shows that the improvements significantly reduces the time required for the verification process or helps the *teacher* to return "*don't know*" in some cases.

Although the one-phase assumption generation method is tested with some initial test systems, there are many things to do for applying the method to the real software industry. We are working on finding out an effective method to divide software systems into components so that the method can be applied. Another work is to find a more effective method to apply the proposed method to systems which have more than two components rather than the heuristic method mentioned in Section 5. This will be the key issue when applying the method in large-scale systems. Finally, we need a kind of graphical user interface for *Tivet* tool that helps software engineers represent their systems and apply the method in practice.

ACKNOWLEDGMENTS

This work has been supported by VNU University of Engineering and Technology under project number CN18.18.

REFERENCES

[1] É. André and S.-W. Lin. Learning-based compositional parameter synthesis for event-recording automata. In A. Bouajjani and A. Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 17–32, Cham, 2017. Springer International Publishing.

[2] Y.-F. Chen, E. M. 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.

[3] E. M. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[4] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[5] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[6] 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.

[7] M. H. de Queiroz, J. E. R. Cury, and W. M. Wonham. Multitasking supervisory control of discrete-event systems. *Discrete Event Dynamic Systems*, 15(4):375–395, Dec 2005.

[8] J. S. Dong, J. Sun, Y. Liu, and Y.-F. Li. Event analytics. In G. Ciobanu and D. Méry, editors, *Theoretical Aspects of Computing – ICTAC 2014*, pages 17–24, Cham, 2014. Springer International Publishing.

[9] J. S. Dong, J. Sun, Y. Liu, Y.-F. Li, J. Sun, and L. Shi. Event and strategy analytics. In *2015 International Symposium on Theoretical Aspects of Software Engineering*, pages 4–6, Sep. 2015.

[10] D. Giannakopoulou, K. S. Namjoshi, and C. S. Păsăreanu. *Compositional Reasoning*, pages 345–383. Springer International Publishing, Cham, 2018.

[11] D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, ASE '02, pages 3–12, Washington, DC, USA, 2002. IEEE Computer Society.

[12] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.

[13] D. Heimbold and D. Luckham. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[14] L. Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency Reflections and Perspectives*, pages 347–374, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[15] C.-L. Le, H.-V. Tran, and P. N. Hung. *On Implementation of the Assumption Generation Method for Component-Based Software Verification*, pages 549–558. Springer International Publishing, Cham, 2017.

[16] Y. Li, Y.-F. Chen, L. Zhang, and D. Liu. A novel learning algorithm for büchi automata based on family of dfas and classification trees. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 208–226, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[17] Y. Li, A. Turrini, Y.-F. Chen, and L. Zhang. *Learning Büchi Automata and Its Applications*, pages 38–98. Springer International Publishing, Cham, 2019.

[18] S.-W. Lin, J. Sun, T. K. Nguyen, Y. Liu, and J. S. Dong. Interpolation guided compositional verification (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 65–74, Nov 2015.

[19] S.-W. Lin, É. André, Y. Liu, J. Sun, and J. S. Dong. Learning assumptions for compositional verification of timed systems. *IEEE Transactions on Software Engineering*, 40(2):137–153, Feb 2014.

[20] S.-W. Lin and P.-A. Hsiung. Compositional synthesis of concurrent systems through causal model checking and learning. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods*, pages 416–431, Cham, 2014. Springer International Publishing.

[21] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley Publishing, 2nd edition, 2006.

[22] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, chapter In Transition from Global to Modular Temporal Reasoning About Programs, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[23] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[24] Y. Sun, G. Lipari, É. André, and L. Fribourg. Toward parametric timed interfaces for real-time components. In *Proceedings 1st International Workshop on Synthesis of Continuous Parameters, SynCoP 2014, Grenoble, France, 6th April 2014.*, volume 145, pages 49–64, 2014.

[25] H.-V. Tran, P. N. Hung, and D. V. Hung. On improvement of assume-guarantee verification method for timed component-based software. In *10th International Conference on Knowledge and Systems Engineering, KSE 2018, Ho Chi Minh City, Vietnam, November 1-3, 2018*, pages 270–275, 2018.