Science of Computer Programming 193 (2020) 102439

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A framework for assume-guarantee regression verification of evolving software

Hoang-Viet Tran [a], Pham Ngoc Hung [a,*], Viet-Ha Nguyen [a], Toshiaki Aoki [b]

[a] University of Engineering and Technology, Vietnam National University, Hanoi, E3 building, 144 Xuan Thuy st. Cau Giay dist. Hanoi, Vietnam
[b] Japan Advanced Institute of Science and Technology, Nomi-shi, 923-1292, Japan

ARTICLE INFO

Article history:
Received 30 May 2019
Received in revised form 16 February 2020
Accepted 29 February 2020
Available online 6 March 2020

Keywords:
Assume-guarantee reasoning
Model checking
Implicit learning
Component-based software evolution
Local weakest assumption

ABSTRACT

This paper presents a framework for verifying evolving component-based software using assume-guarantee logic. The goal is to improve CDNF-based assumption generation method by having local weakest assumptions that can be used more effectively when verifying component-based software in the context of software evolution. For this purpose, we improve the technique for responding to membership queries when generating candidate assumptions. This technique is then integrated into a proposed backtracking algorithm to generate local weakest assumptions. These assumptions are effectively used in rechecking the evolving software by reducing time required for assumption regeneration within the proposed framework. The proposed framework can be applied to verify software that is continually evolving. An implemented tool and experimental results are presented to demonstrate the effectiveness and usefulness of the framework.

1. Introduction

In the last three decades, component-based software engineering (CBSE) has emerged as one of the important approaches in software engineering. This approach has shown a number of advantages such as increasing effectiveness and efficiency, lowering cost, shortening product time-to-market, improving maintainability [52]. As a result, component-based software (CBS) quality assurance plays a critical role in software production life cycles due to the increasing demand for high-quality products. Due to the high-quality standard test procedure in software industry, the verification process in CBSs ensures that certain properties are not violated at all times.

There are two approaches to the verification of modern software: theorem proving which is semi-automatic, requires the interaction of domain experts [21,20,30,37,38,51], and costs a lot of effort [5]; model checking which is automatic and does not require the interaction of domain experts [7,18]. Although the model checking has gained considerable attention due to its fully automatic characteristic, the approach suffers from the problem of *state space explosion* [15,18,48,16]. The assume-guarantee framework [17,19,24,46], which performs modular verification of CBS, has been considered a promising solution for dealing with the state space explosion problem during model checking. The framework uses the "divide-and-conquer" strategy to verify whether a given system satisfies a predefined property. Therefore, it can potentially be applied to large-scale systems in practice. The key problem of the framework is to generate assumptions that satisfy the assume-guarantee rules [19,29,33]. If such an assumption exists, the given system satisfies the required property. Although the framework

* Corresponding author.
E-mail addresses: 15028003@vnu.edu.vn (H.-V. Tran), hungpn@vnu.edu.vn (P.N. Hung), hanv@vnu.edu.vn (V.-H. Nguyen), toshiaki@jaist.ac.jp (T. Aoki).

https://doi.org/10.1016/j.scico.2020.102439
0167-6423/

can be applied to large-scale systems effectively, it does not consider the system under check in the context of software evolution.

Modern software applications are continually evolving, and any verification has to be revisited repeatedly. A reduction in the cost of this repeated verification would offer significant benefits for industry: improving the quality of software through application of verification techniques in situations where this is currently infeasible. Progress has been made using approaches such as labeled transition systems [12,19,31,33–35], implicit representation of transition systems [13,27], timed transition systems [3,28,40–42]. The following two solutions have been used in reducing the verification costs for evolving software.

The first solution is to generate a new assumption each time software evolves at a lower cost. For software modeled by exploiting labeled transition systems, assumptions with small sizes (i.e., assumptions with small numbers of states) can be used effectively to recheck modified software leading to reduced verification cost. In a series of papers, Hung et al. proposed a method to generate minimized assumptions for CBS verification [31,34,35] and a framework to perform modular verification of evolving CBS [33]. However, the cost for generating minimal assumptions can be high [34]. The reason is that the investigated assumption generation problem [19,33–35,31] is formulated as an automata learning problem using the L^* algorithm [4]. As a result, it is difficult to apply this approach to large-scale systems. On the other hand, for the faster assumption generation speed, another verification method, which uses CDNF (Conjunction of Disjunctive Normal Form) algorithm [10] and implicit representation of software, was proposed in 2010 by Chen et al. [13]. Later, in 2016, this method was improved by He et al. and applied in CBS regression verification [26] by introducing a fine–grained learning technique. However, with modified software, some of the subpredicates of the new version of components can be different, which requires the regression verification progress to regenerate the assumptions for every small change in the software component.

The second solution to reduce the verification cost for modified software is to increase assumption reuse as much as possible. This is because the software development cycle involves daily change. Therefore, the less time required to regenerate assumptions, the greater the cost savings when verifying modified software. Moreover, from the analysis in Section 5 below, weak assumptions (i.e., assumptions with large languages) can help to achieve this purpose and play a key role in the verification of modified software. On the other hand, to our knowledge, no research has been conducted on generating assumptions that have the weakest languages and use implicit specification. As a result, this research focuses on improving the learning algorithm proposed by Chen [13] to generate local weakest assumptions that can be used more efficiently to reduce the cost of software regression verification during software evolution.

To achieve the above goal, we first improve the technique to answer membership queries for the two ι (i.e., the initial predicate) and τ (i.e., transition relation) CDNF learning instances. Based on this improved answering technique, we can generate weaker assumptions than those generated by the algorithm proposed by Chen et al. [13] (hereafter, we refer to as CBAG algorithm) using a proposed backtracking learning algorithm (referred to as LWAG algorithm). This leads to an important result in the context of software evolution: LWAG algorithm can reduce the number of times assumptions must be regenerated when verifying modified software. The improved answering technique and LWAG algorithm are integrated into a framework to effectively reduce the number of times assumption regeneration is required for evolving software.

Using assumption generation algorithms which employ the implicit representation, we can not only benefit from the fast learning process but we can also obtain several advantages of implicit software representation over explicit representation. First, the contextual assumptions represented implicitly using Boolean functions have fewer states than do assumptions modeled using deterministic finite automata because implicit representations are equivalent to nondeterministic finite automata, which are exponentially more succinct than deterministic ones. As a result, our generated assumptions can have an exponentially smaller number of states than do assumptions generated from explicit representations. The second advantage is the scalability of the verification method using implicit representations, which occurs because the L^* algorithm requires a polynomial number of queries in the number of states of the target finite automaton [4,49]. In contrast, the CDNF algorithm requires a polynomial number of queries in the number of Boolean variables of the target Boolean function [10]. Because implicit assumptions can be exponentially more succinct than explicit ones, the learning algorithms for implicit assumptions can be exponentially better than automata-theoretic ones.

To our knowledge, the first paper that proposed using the L^* algorithm to learn assumptions for the assume-guarantee reasoning algorithm was Cobleigh et al. [19]. Following this paper, several studies improved the method, including adoption of the assume-guarantee rules [6,26,39,45], symbolic implementation for assume-guarantee rules [8,9,45], several improvements proposed in [1,2,12,14,25,50,53], and an extension to support liveness properties [22]. However, these papers all use the L^* algorithm to learn an automaton as the required contextual assumption. Hence, they all have the same disadvantages as described above compared to the algorithm proposed in Chen's paper [13]. Hence, we based our paper on Chen's algorithm [13] to verify modified software.

The remainder of this paper is organized as follows. Section 2 presents the background for this paper. We review CBAG algorithm for generating assumptions using the CDNF algorithm in Section 3, followed by the proposed algorithms to improve the answers to membership queries and generate assumptions in Section 4. Section 5 presents a framework for verifying modified CBSs using assumptions generated by the proposed learning algorithm. Section 6 shows the preliminary experimental results. Related papers are presented in Section 7. Finally, we conclude the paper in Section 8.

Table 1
Valuation functions for trace example.

Valuation	x_1	x_2	x_3	x_4
v_0	F	F	-	-
v_1	F	T	F	T
v_2	T	F	T	F

2. Background

In this section, we present some basic concepts used in this paper. We use $\mathbb{B}$ to denote the Boolean domain, which is a set that consists of exactly two elements whose interpretations are T (*true*) and F (*false*) (i.e., $\mathbb{B} = \{T, F\}$). Given a set of Boolean variables X, we call |X| the size of X, where |X| is the number of variables inside X.

Let X be a finite set of Boolean variables. Consider a function $\theta(X)$ over X, which is a function from $\mathbb{B}^{|X|}$ to the Boolean domain $\mathbb{B}$, $\theta(X)$ is called a *Boolean function*. Let $v : X \to \mathbb{B}$ be a function over X that maps each $x \in X$ to one value in $\mathbb{B}$. We call v a *valuation* of X. The result of evaluating ϕ by replacing each $x \in X$ with $v(x)$ is denoted by $\phi[v]$. We use "|", "$\wedge$", and "$\neg$" to denote the logical OR, AND, and NOT operators, respectively. Let consider an example where $\phi(X) = \neg x_1 \wedge \neg x_2$, in which $X = \{x_1, x_2\}$. If v is a *valuation* where $v(x_1) = T$ and $v(x_2) = T$, then $\phi[v] = \neg T \wedge \neg T = F \wedge F = F$.

Consider a set of Boolean variables $Y \subseteq X$, we call $v \restriction_Y$ the *restriction* of v on Y. That is, $v \restriction_Y : Y \to \mathbb{B}$ and $v \restriction_Y (y) = v(y)$ for every $y \in Y$. As a result, for a finite sequence of valuations $\alpha = v^0 v^1 \ldots v^t$ over X and $Y \subseteq X$, we call $\alpha \restriction_Y = v^0 \restriction_Y v^1 \restriction_Y \ldots v^t \restriction_Y$ the *restriction* of α on Y. For the above example where $X = \{x_1, x_2\}$, if v is a *valuation* where $v(x_1) = T$, $v(x_2) = T$ and $Y = \{x_2\}$, then $v \restriction_Y (x_2) = v(x_2) = T$.

Definition 1 *(Transition system).* A *transition system* M is a 3-tuple $\langle X, \iota(X), \tau(X, X')\rangle$, where X is a finite set of Boolean variables, $\iota(X)$ is a Boolean function (called the *initial predicate*) and $\tau(X, X')$ is also a Boolean function (called a *transition relation*). X′ is described as follows.

Let X be a finite set of Boolean variables. The *next state* of X is also a finite set of Boolean variables $X' = \{x' : x \in X\}$.

Let $\psi(X, X')$ be a Boolean function over X and X′ and v and v' be two valuations over X. The result of evaluating ψ by replacing each $x \in X$ with $v(x)$ and each $x' \in X'$ with $v'(x)$ is denoted by $\psi[v, v'])$. Consider the case where $\psi(X, X') = \neg x_1 \wedge x_2 \wedge x_3 \wedge \neg x_4$, $X = \{x_1, x_2\}$ and $X' = \{x_3, x_4\}$. If v and v' are the two *valuations* where $v(x_1) = T$, $v(x_2) = F$, $v'(x_3) = T$, and $v'(x_4) = T$, then $\psi[v, v'] = \neg T \wedge F \wedge T \wedge \neg T = F \wedge F \wedge T \wedge F = F$.

A finite sequence of valuations $\alpha = v^0 v^1 \ldots v^t$ is called a *trace* of M if and only if v^i is a valuation over X such that $\iota[v^0] = T$ and $\tau[v^i, v^{i+1}] = T$ for $0 \leq i < t$. The number of valuations in α is called the *length* of α, denoted by $|\alpha|$. The set of all traces of M is called the language of M and denoted by $L(M)$.

Consider a transition system $M = \langle X, \iota(X), \tau(X, X')\rangle$, where $X = \{x_1, x_2\}$, $X' = \{x_3, x_4\}$, $\iota = (\neg x_1 \wedge \neg x_2)$, $\tau = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge x_4)|(\neg x_1 \wedge x_2 \wedge x_3 \wedge \neg x_4)$. Consider a trace $v = v_0 v_1 v_2$, where v_0, v_1, and v_2 are defined in Table 1. Using these valuation functions, we can see that the trace $v = v_0 v_1 v_2$ is a trace of M because $\iota[v_0] = \neg F \wedge \neg F = T \wedge T = T$; $\tau[v_0, v_1] = (\neg F \wedge \neg F \wedge \neg F \wedge T)|(\neg F \wedge F \wedge F \wedge \neg T) = T$; and $\tau[v_1, v_2] = (\neg F \wedge \neg T \wedge \neg T \wedge F)|(\neg F \wedge T \wedge T \wedge \neg F) = T$.

Definition 2 *(Satisfiability).* Consider a transition system $M = \langle X, \iota(X), \tau(X, X')\rangle$ and a *state predicate* $\pi(X)$, which is a Boolean function over X. We say that M *satisfies* π (denoted by $M \models \pi$) if and only if $\forall \alpha = v^0 v^1 \ldots v^t \in L(M)$, we have $\pi[v^i] = T$ for $0 \leq i \leq t$.

Consider the example system M mentioned in Definition 1 when the *state predicate* is defined as follows: $\pi(X) = (\neg x_1 \wedge \neg x_2)|(\neg x_1 \wedge x_2)|(x_1 \wedge \neg x_2)$. We can see that $\forall v_0 : \iota[v_0] = T$, we have $\pi[v_0] = T$. In addition, $\forall v_i, v_{i+1} : \tau[v_i, v_{i+1}] = T$, and we also have $\pi[v_i] = T$ and $\pi[v_{i+1}] = T$. Therefore, we have $M \models \pi$.

Let M be a transition system and π be a state predicate, the problem of deciding whether M satisfies π is called the *invariant checking* problem. The technique for solving the *invariant checking* problem automatically is called a *model check*. When performing a model check for $M \models \pi$, a model checking algorithm returns a witness if M does not satisfy π. A trace $v^0 v^1 \ldots v^t$ of M is called a *witness* to $M \not\models \pi$ if and only if $\pi[v^i] = T$ for $0 \leq i < t$ but $\pi[v^t] = F$.

Definition 3 *(Simulation).* Let $N = \langle X, \iota_N(X), \tau_N(X, X')\rangle$ be a transition system. We say that N *simulates* M or M *is simulated by* N (denoted by $M \preceq N$) if $\forall X.\iota_M(X) \Rightarrow \iota_N(X)$ and $\forall XX'.\tau_M(X, X') \Rightarrow \tau_N(X, X')$.

Intuitively, if the initial condition of M is more restrictive than that of N and all transitions allowed in M are also allowed in N, then N simulates M. Formally, if $M \prec N$, then $L(M) \subset L(N)$. In this case, we say that M is *stronger* than N or N is *weaker* than M.

Composition is one of the key operations during the assume-guarantee verification process of a CBS. It describes the behavior of a CBS from its sub-components.

Definition 4 *(Composition).* Let $M_i = \langle X_i, \iota_i(X_i), \tau_i(X_i, X_i')\rangle$ be a transition system over a set of Boolean variables X_i for i = 0,1. The *composition* of M_0 and M_1 is the transition system $M_0||M_1 = \langle X_0 \cup X_1, \iota_0(X_0) \wedge \iota_1(X_1), \tau_0(X_0, X_0') \wedge \tau_1(X_1, X_1')\rangle$.

For any finite sequence of valuations α over $X_0 \cup X_1$, $\alpha \in L(M_0||M_1)$ if and only if $\alpha \restriction_{X_0} \in L(M_0)$ and $\alpha \restriction_{X_1} \in L(M_1)$. Let consider the following example of the composition operation. In this paper's examples, we use j to denote x_j, $-j$ to denote $\neg x_j$, 0 to denote the Boolean value F, and 1 to denote the Boolean value T. Let $M = M_0 \parallel M_1$ be a CBS, where M_0 and M_1 are defined as follows:
$X_{M_0} = \{1, 2\}$,
$X'_{M_0} = \{3, 4\}$,
$\iota_{M_0} = (-1 \wedge -2)$,
$\tau_{M_0} = (-1 \wedge -2 \wedge -3 \wedge 4)|(-1 \wedge 2 \wedge 3 \wedge -4)$, and
$X_{M_1} = \{5, 6\}$,
$X'_{M_1} = \{7, 8\}$,
$\iota_{M_1} = (-5 \wedge -6)$,
$\tau_{M_1} = (-5 \wedge -6 \wedge -7 \wedge 8)|(-5 \wedge 6 \wedge 7 \wedge -8)$
From Definition 4, we have M defined as follows:
$X_M = \{1, 2, 5, 6\}$,
$X'_M = \{3, 4, 7, 8\}$,
$\iota_M = (-1 \wedge -2) \wedge (-5 \wedge -6)$, and
$\tau_M = ((-1 \wedge -2 \wedge -3 \wedge 4)|(-1 \wedge 2 \wedge 3 \wedge -4)) \wedge ((-5 \wedge -6 \wedge -7 \wedge 8)|(-5 \wedge 6 \wedge 7 \wedge -8))$

Definition 5 *(Noncircular Assume-guarantee rule [13]).* Let $M_i = \langle X_i, \iota_i(X_i), \tau(X_i, X_i')\rangle$ be a transition system for i = 0,1 and π be a state predicate over $X_0 \cup X_1$. The following formula is called the assume-guarantee rule.

$$\frac{M_0||A \vDash \pi \qquad M_1 \preceq A}{M_0||M_1 \vDash \pi}$$

where $A = \langle X_1, \iota_A(X_1), \tau_A(X_1, X_1')\rangle$ is a transition system, $M_0||A \vDash \pi$ and $M_1 \preceq A$ are its premises, and $M_0||M_1 \vDash \pi$ is its conclusion. The assume-guarantee rule is *sound* and *invertible*. That means its conclusion holds if and only if its premises are fulfilled.

Definition 6 *(Assumption).* Let $M_i = \langle X_i, \iota_i(X_i), \tau_i(X_i, X_i')\rangle$ be a transition system for i = 0,1 and π be a state predicate over $X_0 \cup X_1$. Let A be a transition system in which $A = \langle X_1, \iota_A(X_1), \tau_A(X_1, X_1')\rangle$. If $M_0||A \vDash \pi$ and $M_1 \preceq A$, then A is called the *contextual assumption* of M_0. We will hereafter call A the *assumption*.

Definition 7 *(Weakest assumption).* Among all assumptions that satisfy Definition 6, the assumption A_W is called the *weakest assumption* if and only if $\forall A : L(A) \subseteq L(A_W)$.

To the best of our knowledge, there has not been any algorithm which can generate the weakest assumption. For a given CBS $M = M_0 \parallel M_1$ and a predefined property π, it could be verified that if a given trace σ belongs to $L(A_W)$ by using a membership query (shown in Section 3.2.1). A_W is known as an assumption whose language is the set of traces that the corresponding membership queries results are *yes*.

Remark 1. Let $\mathfrak{A}$ be a subset of assumptions that satisfy Definition 6. We call $A_{LW} \in \mathfrak{A}$ the *local weakest assumption* in $\mathfrak{A}$ if and only if $\forall A \in \mathfrak{A} : L(A) \subseteq L(A_{LW})$.

3. The CDNF–based assumption generation method

3.1. The CDNF algorithm

Let X be a fixed set of Boolean variables and $\lambda(X)$ be a Boolean function over X. CDNF is an incremental learning algorithm that can learn the exact representation of $\lambda(X)$ in a finite number of steps [10]. Sharing the same ideas as the L^* algorithm [4], CDNF is based on a *teacher* (which knows $\lambda(X)$) when performing the learning process. The *teacher* must be able to answer the following two types of queries:

- *Membership queries* $MEM(v)$: Given a valuation v over X, if $\lambda[v] = T(true)$, the *teacher* returns *yes* to the *learner*. Otherwise, it returns *no*.
- *Equivalence queries* $EQ(h)$: Given a candidate Boolean function h over X, if the candidate h is equivalent to the target function λ, the *teacher* returns *yes*. Otherwise, the *teacher* returns a valuation of v over X such that $h[v] \neq \lambda[v]$. The valuation v serves as a counterexample to the equivalence query.

Consider an example where $\lambda(x, y) = (\neg x \wedge \neg y) \vee (x \wedge \neg y)$ is the target Boolean function over x and y. The *teacher* returns *no* to the membership query $MEM(v)$, where $v(x) = F$, and $v(y) = T$ (denoted by $v(xy) = FT$) because $\lambda(FT) = F$. With another valuation $v(xy) = FF$, the *teacher* answers *yes*. Later, the *learner* generates a candidate $h((x, y)) = \neg x \wedge \neg y$ and sends an equivalence query $EQ(h)$ to the *teacher* for the candidate $h(x, y) = \neg x \wedge \neg y$. The *teacher* provides the valuation $v(xy) = TF$ as a counterexample to the *leaner* because $h[v] = F \neq T = \lambda[v]$. Based on this counterexample, the *learner* generates another candidate $h'(x, y) = (\neg x \wedge \neg y) \vee (x \wedge \neg y)$ and then sends a new equivalence query $EQ(h')$ to the *teacher*. This time, the *teacher* returns *yes* to the *learner* and the learning process stops.

Consider a Boolean function of $\lambda(X)$ over X. Let $|\lambda(X)|_{DNF}$ and $|\lambda(X)|_{CNF}$ be the corresponding size of $\lambda(X)$ in the *minimal* disjunctive and conjunctive normal forms, respectively. The CDNF algorithm can learn representations of any target Boolean function in a polynomial number of queries in $|\lambda(X)|_{DNF}$, $|\lambda(X)|_{CNF}$, and $|X|$ [10].

3.2. The CDNF–based assumption generation algorithm

This section presents the CDNF–Based assumption generation algorithm [13], referred to as CBAG algorithm. Based on this algorithm, the LWAG algorithm is presented in Section 4. We start with some core algorithms that will be integrated in CBAG algorithm to generate assumptions [13]. These algorithms are membership query answering algorithm (Algorithm 1 - OMQ algorithm), equivalence query answering algorithm (Algorithm 2 - EQ algorithm), and an algorithm that checks whether a counterexample α can be used to learn a better candidate assumption or if α is a real counterexample to the fact that $M_0 \parallel M_1 \not\models \pi$ (Algorithm 3 - IW algorithm). The correctness of these algorithms was proved by Chen et al. [13].

3.2.1. The original membership query answering algorithm

This section presents the OMQ algorithm to resolve the membership queries for both the $CDNF_\iota$ and $CDNF_\tau$ learning instances [13]. The pseudo code for the algorithm is shown in Algorithm 1, where the input is a $type$ parameter (which can be either ι or τ) and a valuation v (which can be either one valuation μ or a pair of valuations (μ, μ') respectively) as input. When the $type = \iota$, then OMQ algorithm will check whether $\theta_1 = \iota_1(\mu) = T$ (line 1); otherwise, it checks whether

Algorithm 1 (OMQ algorithm) $IsMember(type, v)$.

Input: (ι, μ): a membership query for the target $\iota_A(X)$; or $(\tau, (\mu, \mu'))$: a membership query for the target $\tau_A(X_1, X_1')$
Output: *yes* or *no*

```
1: if θ1(v) = T then                ▷ When type is ι, θ1 is ι1; when type is τ, θ1 is τ1
2:     return yes
3: else
4:     return no
5: end if
```

$\theta_1 = \tau_1(\mu, \mu') = T$ (line 1). When $\theta_1(v) = T$, OMQ algorithm returns *yes* to the *learner*; otherwise, it returns *no*.

3.2.2. The original equivalence query answering algorithm

When both $CDNF$ instances implemented in the *learner* have their own candidate functions ι and τ, the *learner* will send an equivalence query to the *teacher*. The EQ algorithm presented in Algorithm 2 is implemented in the *teacher* to answer the equivalence query from the *learner*. EQ algorithm starts by constructing the candidate assump-

Algorithm 2 (EQ algorithm) $IsEquivalent(\iota, \tau)$.

Input: $EQ(\iota)$: an equivalence query for the target $\iota_A(X_1)$; $EQ(\tau)$: an equivalence query for the target $\tau_A(X_1, X_1')$;
Output: *yes*, or *continue* and a counterexample to $EQ(\iota)$, or *continue* and a counterexample to $EQ(\tau)$

```
1: Let C be the transition system (X1, ι1(X1), τ1(X1, X1'));
2: if ι1(X1) ∧ ¬ι(X1) is satisfied by μ then
3:     Answer EQ(ι) with the counterexample μ;
4:     return continue;
5: end if
6: if τ1(X1, X1') ∧ ¬τ(X1, X1') is satisfied by μμ' then
7:     Answer EQ(τ) with the counterexample μμ';
8:     return continue;
9: end if
10: if M0 ∥ C ⊨ π then
11:     Answer EQ(ι) with yes;
12:     Answer EQ(τ) with yes;
13:     return yes and report "M0 ∥ M1 ⊨ π";
14: else
15:     Let α be a witness to M0 ∥ C ⊭ π;
16:     Call IsWitness(α);
17: end if
```

tion $C = (X_1, \iota_1(X_1), \tau_1(X_1, X_1'))$ in line 1. It then checks whether a valuation μ exists in which $\iota_1[\mu] \wedge \neg\iota[\mu] = T$ (line 2).

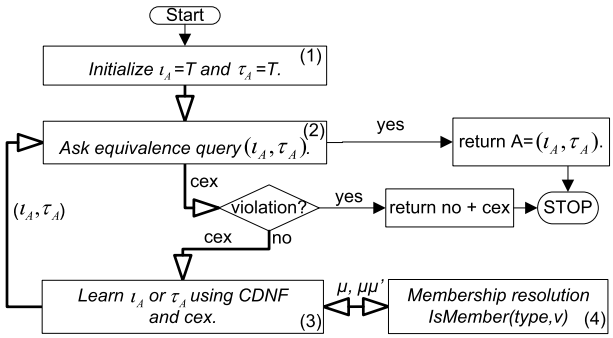

Fig. 1. CBAG algorithm.

If so, then μ is returned to the $CDNF_\iota$ instance as a counterexample for it to learn a new candidate initial function ι' (line 3) and *continue* is returned to the *learner* (line 4). When there is a candidate function ι for which there is no valuation μ that $\iota_1[\mu] \wedge \neg\iota[\mu] = T$, the candidate function of ι is a satisfied initial function for ι_A. Next, the algorithm checks whether a pair (μ, μ') exists in which $\tau_1[\mu, \mu'] \wedge \neg\tau[\mu, \mu'] = T$ (line 6). If the answer is yes, the algorithm returns (μ, μ') as a counterexample for the $CDNF_\tau$ instance to learn another candidate function τ' (line 7) and *continue* to the *learner* (line 8). Otherwise, the candidate function of τ is a satisfied transition function for τ_A. The last step is to check whether the candidate assumption C satisfies the Assume-Guarantee rule in Definition 5 (line 10). If so, the algorithm returns both $CDNF$ instances with *yes* (lines 11 and 12), returns *yes* to the *learner*, and reports "$M_0 \parallel M_1 \models \pi$" (lines 13). Otherwise, let α be the witness to $M_0 \parallel C \not\models \pi$; the algorithm calls $IsWitness(\alpha)$ (i.e., IW algorithm) to check whether α can be the counterexample for ι, τ; otherwise, it witnesses the fact that $M_0 \parallel M_1 \not\models \pi$.

3.2.3. The original witness analysis algorithm

When $M_0 \parallel C \not\models \pi$ is witnessed by a counterexample α, it is required IW algorithm (presented in Algorithm 3) to analyze whether α can be returned to either $CDNF_\iota$, $CDNF_\tau$ or whether it actually witnesses that $M_0 \parallel M_1 \not\models \pi$. IW algorithm

Algorithm 3 (IW algorithm) $IsWitness(\alpha)$.

Input: α is a witness to $M_0 \parallel C \not\models \pi$
Output: *continue* and a counterexample to $EQ(\iota)$, or *continue* and a counterexample to $EQ(\tau)$, or *no* to the *learner* and a counterexample

```
1: Let α ↾_{X_1} = μ^0 μ^1 ... μ^t;
2: if ι_1[μ^0] = F then
3:     Answer EQ(ι) with the counterexample μ^0;
4:     return continue;
5: end if
6: for i := 1 to t do
7:     if τ_1[μ^{i-1}, μ^i] = F then
8:         Answer EQ(τ) with the counterexample μ^{i-1} μ^i;
9:         return continue;
10:    end if
11: end for
12: return no + α and report "M_0 ∥ M_1 ⊭ π is witnessed by α";
```

starts by restricting α on X_1 in line 1. Let $\mu^0\mu^1...\mu^t$ be the result. Then, it checks whether $\iota_1[\mu^0] = F$, and finally, it returns μ^0 to the $CDNF_\iota$ so that this instance can learn another better candidate initial function ι' (line 3) and *continue* to the *learner* (line 4). When $\iota_1[\mu^0] = T$, the algorithm continues to find a couple of valuations $\mu^{i-1}, \mu^i \in \{\mu^0, \mu^1, ..., \mu^t\}$, such that $\tau_1[\mu^{i-1}, \mu^i] = F$ (line 7). This pair (μ^{i-1}, μ^i) will be returned to the $CDNF_\tau$ so that this instance can learn another, better transition function τ' (line 8) and *continue* to the *learner* (line 9). When no such couple (μ, μ') exists, the algorithm returns $no + \alpha$ and reports that "$M_0 \parallel M_1 \not\models \pi$ is witnessed by α" and stops (line 12).

3.2.4. The original assumption generation algorithm

The CBAG algorithm creates a *learner* with two instances of the CDNF algorithm [10], called instances $CDNF_\iota$ and $CDNF_\tau$. These instances interact with a *teacher*, which is where OMQ, EQ, and IW algorithms are implemented. An overview of CBAG algorithm is shown in Fig. 1. Flowcharts are used to present both CBAG and LWAG algorithms because of the big complexity of these algorithms which contain both $CDNF_\iota$ and $CDNF_\tau$ instances with other assumptions related computation. Note that we use the notation $A = (\iota_A, \tau_A)$ as a brief representation of the contextual assumption to be generated as defined in Definition 6. Hereafter, we also use arrows with empty head (⟶▷) to show data-flow, and arrows with solid head (⟶▶) to show control-flow.

In CBAG algorithm, the two Boolean functions of ι_A and τ_A are initialized in step 1 with T (*true*). For each of the *conjectures* (ι_A, τ_A) (i.e., an assumption candidate), the *learner* sends the *teacher* an *equivalence query* in step 2. EQ

Table 2
Implication operation truth table.

θ	γ	$\theta \rightarrow \gamma$
F	F	T
T	F	F
F	T	T
T	T	T

algorithm is implemented in this step in the *teacher* to answer the *learner*. If the *teacher* returns *yes*, the algorithm stops and returns the *conjecture* (ι_A, τ_A) as the needed assumption. If the *teacher* returns *no* and a counterexample (*cex*) after analyzing that the given system violates the property, the algorithm also stops and returns *no* and *cex*. This answer means that the given system violates the property with a counterexample *cex*. After analyzing in IW algorithm that using the counterexample *cex* can generate another candidate function, if the *teacher* returns *continue* and a counterexample *cex* to either $CDNF_\iota$ or $CDNF_\tau$, the *learner* will use *cex* to learn a new corresponding candidate function (either ι_A or τ_A) (step 3). In this step, the *learner* interacts with the *teacher* which uses OMQ algorithm to answer the *learner* (step 4). When it finishes learning and creating a new *conjecture*, the *learner* will ask a new *equivalence query* by returning to step 2. This loop (from step 2 to 4) is repeated until the *teacher* returns either *yes* or *no and cex*.

Although CBAG algorithm can nicely generate assumptions, it does not support system verification in the context of software evolution. If we simply use the method to generate assumptions as described in the framework proposed by Hung et al. [33], there will be no reduction in the number of times assumptions need to be regenerated. Consequently, during system change, this process involves significant effort when rechecking modified systems as they evolve daily during their development cycle. The sections below describe a way to reduce the number of times assumptions must be regenerated by generating weaker assumptions than those generated by CBAG algorithm and integrating those assumptions into a framework for rechecking modified systems.

4. A local weakest assumption generation method

4.1. An improved technique for answering membership queries

As shown above in Section 3.1, in CDNF algorithm, the generated Boolean function depends on how the *teacher* answers membership queries and whether *yes* or *no* (i.e., $\lambda[v] = T$ or $\lambda[v] = F$, respectively) are returned to the *learner*. As a result, to improve the CDNF–based assumption generation method, we first need to focus on improving the technique by which of the *teacher* answers the *learner*.

After analyzing OMQ algorithm together with Table 2, we observe that the answering technique in this algorithm can be improved as follows. The relationship between M_1 and A in Definition 6 implies that $\iota_{M_1}(X_1) \rightarrow \iota_A(X_1)$ and $\tau_{M_1}(X_1, X_1') \rightarrow \tau_A(X_1, X_1')$. Table 2 shows a truth table of the implications of the Boolean operation, where θ and γ are two arbitrary Boolean functions. From Table 2, we can see that the technique to answer membership queries in Algorithm 1 is correct, but does not cover all the cases where the answer can be *yes*. To find that $\theta \rightarrow \gamma = T$, a result of $\theta = T$ (*true*) guarantees that $\gamma = T$; however when $\theta = F$ (*false*), there is still a case where $\gamma = T$. Based on this observation, an improved version of OMQ algorithm is presented in Algorithm 4 - IMQ algorithm. A new symbol, *question*, is returned to the *learner* when $\theta = F$, whereas θ is either $\iota_1(X_1)$ or $\tau_1(X_1, X_1')$ (line 4). The *learner* will first make a copy of the learning status before

Algorithm 4 (IMQ algorithm) *ImprovedIsMember*(*type*, *v*).

Input: (ι, μ): a membership query for the target $\iota_A(X)$; or $(\tau, (\mu, \mu'))$: a membership query for the target $\tau_A(X_1, X_1')$
Output: *yes* or *question*

```
1: if θ[v] = T then                          ▷ When type is ι, θ is ι1; when type is τ, θ1 is τ1
2:     return yes
3: else
4:     return question
5: end if
```

treating the *question* results as a *yes* to generate the candidate function. Then, it sends *equivalence queries* to the *teacher*. When the candidate does not satisfy the assume-guarantee rule in Definition 5, it retrieves the previously stored learning status and treats the corresponding *question* result as a *no*, at which point it starts the learning process again. However, when $\theta = T$, the algorithm returns *yes* to the *learner* in the same way as OMQ algorithm (line 2).

4.2. A backtracking local weakest assumption generation algorithm

Using the improved answering technique to membership queries in IMQ algorithm, this section shows a backtracking algorithm, known as LWAG algorithm, that generates weaker assumptions than those generated by CBAG algorithm shown in Section 3. LWAG algorithm is shown in Fig. 2. A correctness proof of LWAG algorithm will be presented in Section 4.3.

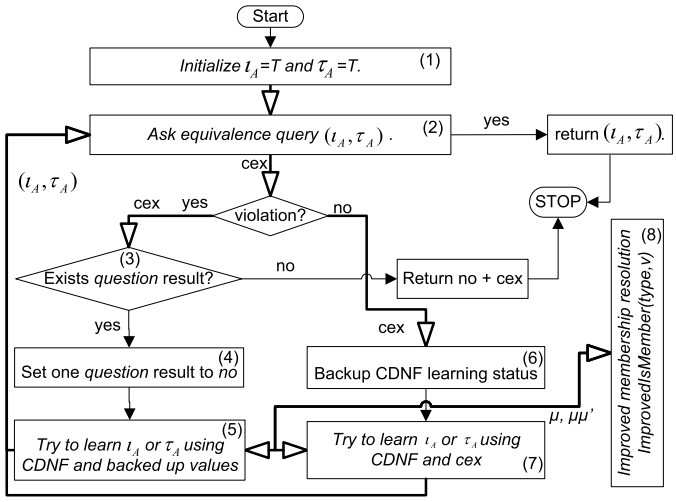

Fig. 2. LWAG algorithm to generate local weakest assumptions for CBSs.

The key idea of LWAG algorithm is to perform a "try" operation to learn either an initial or transition Boolean function (ι_A or τ_A) for each *question* membership query result from the *teacher*. Initially, the *question* result is considered as a "*yes*". When an attempt is unsuccessful, the algorithm backtracks one step and considers the *question* result as a "*no*". The algorithm then restarts the learning process with the newly considered as "*no*" membership query result. To be able to backtrack one step after an unsuccessful attempt, we need to make a copy of the learning status of the CDNF algorithm before treating the *question* result as "*yes*", allowing it to backtrack and restore the learning process later.

In LWAG algorithm, the *learner* initializes the initial function ι_A and transition function τ_A of the candidate assumption (*conjecture*) to be generated with T (*true*) in step 1. Subsequently, for each *conjecture* of (ι_A, τ_A), the *learner* sends the *teacher* an *equivalence query* (step 2). The *teacher* uses EQ algorithm to check whether the *conjecture* satisfies the assume-guarantee rules in Definition 5. If the *teacher* returns a *yes* answer, the algorithm terminates and returns *yes* and the *conjecture* (ι_A, τ_A) as the needed assumption. If the *teacher* returns *continue* and *cex* after the analysis, using the *cex*, the *learner* will generate a new candidate Boolean function. Depending on whether *cex* is for $CDNF_\iota$ or $CDNF_\tau$, the *learner* will make a copy of the status of the corresponding learning process (i.e., the CDNF algorithm's status for learning ι_A or τ_A) (step 6) before learning a new function (ι_A or τ_A) (*conjecture* function) for the next *conjecture* (step 7). In step 7, the *learner* interacts with the *teacher* which is using IMQ algorithm while "walking" (step 8) to learn a new *conjecture* function. During walking, the *learner* first treats a *question* result as a *yes* to generate a *conjecture* function. The *learner* also stores membership query results returned from the *teacher* in a list that can be checked later. When the *learner* finishes learning a new *conjecture*, it comes back to step 2 to ask the *teacher* the corresponding *equivalence query*. If the *teacher* returns *no* and a counterexample *cex* after determining that the given system violates the given property, then the *learner* checks whether any membership query result exists with a *question* result (step 3). If *yes*, this means that the corresponding valuation is not a member of the target *conjecture* function. The *learner* will change the corresponding *question* to *no* (step 4), return to the previous step and start the learning process again from the backed-up status by considering the *question* result as a *yes* (step 5). Note that the *learner* will still interact with the *teacher* by sending membership queries while walking (step 8), but it will not repeat the membership queries for valuations that have already been asked. For each a new *conjecture*, the *learner* will return to step 2 to ask a new *equivalence query* for the newly created conjecture. If the *teacher* returns *no* and *cex* after determining that the given system violates π, but no *membership query* result of *question* exists in the list of results, then all the possible cases have been tried in which a valuation μ (or $\mu\mu'$) is such that when $\iota_1[\mu] = F$ (or $\tau_1[\mu, \mu'] = F$), the corresponding $\iota_A[\mu] = T$ (or $\tau_A[\mu, \mu'] = T$), but without success. Consequently, no suitable assumption can be found for the given M_0, M_1, and π. At that point, the algorithm returns *no and cex* and terminates.

4.3. Correctness

LWAG algorithm was developed based on CBAG algorithm, in which each of the learning steps is actually an attempt to learn either an initial or transition Boolean function (ι_A or τ_A). When an attempt does not successfully generate a satisfied *conjecture*, the *learner* backtracks to the step before the attempt to learn a new *conjecture*, which is why it is called a *backtracking* algorithm. Each attempt actually contains a part of CBAG algorithm – the learning process that starts at step 4 of LWAG algorithm. The algorithm changes one *question* result to *no* and learns a new *conjecture*, which it then uses to send the *teacher* the first equivalence query, and so on.

To prove the correctness of LWAG algorithm, we follow the three steps below to prove its soundness, completeness, and termination. Let $M_i = \langle X_i, \iota_i(X_i), \tau_i(X_i, X_i')\rangle$ be transition systems for $i = 0, 1$ and π be a state predicate over $X = X_0 \cup X_1$. We have the following lemmas.

Lemma 1 *(Soundness).*

1. *Let $\iota(X_1)$ and $\tau(X_1, X_1')$ be Boolean functions over X_1 and $X_1 \cup X_1'$, respectively. If LWAG algorithm reports yes, then $M_0||M_1 \models \pi$ and $A = \langle X_1, \iota(X_1), \tau(X_1, X_1')\rangle$ is the corresponding assumption.*
2. *Let $\iota(X_1)$ and $\tau(X_1, X_1')$ be Boolean functions over X_1 and $X_1 \cup X_1'$, respectively. If LWAG algorithm reports no and cex, then that cex is the witness to $M_0||M_1 \not\models \pi$.*

Proof. When LWAG algorithm reports *yes* in step 2, it has verified that the conjecture $A = \langle X_1, \iota_A(X_1), \tau_A(X_1, X_1')\rangle$ is actually a required assumption using EQ algorithm (step 2). Based on the correctness of that algorithm, we have $M_0||M_1 \models \pi$. In contrast, when the algorithm reports *no and cex* (step 3 returns *no*), it has verified that the list of membership query results contains no *question* result, which means there is no difference between the results returned by OMQ algorithm and the results returned by IMQ algorithm. Therefore, the same *conjecture* as in CBAG algorithm has been submitted to EQ algorithm for an equivalence query. Consequently, the answer *no and cex* is correct and the *cex* is the witness to $M_0||A \not\models \pi$ by the correctness of EQ algorithm. When the *teacher* returns *continue* and a counterexample *cex* that the *learner* can use to generate a new Boolean function candidate after analyzing in IW algorithm, the algorithm continues creating new *conjecture* functions and submitting them to the *teacher* (steps 6 and 7). The algorithm terminates in the two cases described above, where the *teacher* returns either *yes* or *no and cex* and there is no *question* result in the list of membership query results. □

Lemma 2 *(Completeness).*

1. *If $M_0||M_1 \models \pi$, then LWAG algorithm reports yes for some Boolean functions $\iota(X_1)$ and $\tau(X_1, X_1')$ over X_1 and $X_1 \cup X_1'$, respectively.*
2. *If cex is a witness to $M_0||M_1 \not\models \pi$, then LWAG algorithm reports no and cex for some Boolean functions $\iota(X_1)$ and $\tau(X_1, X_1')$ over X_1 and $X_1 \cup X_1'$, respectively.*

Proof. When $M_0||M_1 \models \pi$, based on the correctness of CBAG algorithm, there exists a $\iota(X_1)$ and $\tau(X_1, X_1')$ such that a conjecture of $A = \langle X_1, \iota(X_1), \tau(X_1, X_1')\rangle$ satisfies the rule in Definition 6. Because EQ algorithm returns *yes*, LWAG algorithm also returns *yes* (step 2). In contrast, when *cex* is a witness to $M_0||M_1 \not\models \pi$, we consider the following two cases. When some *question* results exist in the membership query result list (i.e., step 3 returns *yes*), the algorithm will not return the verification result yet. Instead, it continues setting one of the *question* results to *no* (step 4) and trying to learn new conjecture functions (step 5). This process repeats until no *question* result remains in the list. In this case, the conjecture submitted to EQ algorithm is the same as in CBAG algorithm. If the algorithm still returns *no and cex*, this response is equivalent to returning a witness of *cex* to the original learner to witness the fact that $M_0||M_1 \not\models \pi$ (step 3 returns *no*). In that case, LWAG algorithm returns *no and cex* for the conjecture functions of $\iota(X_1)$ and $\tau(X_1, X_1')$. Alternatively, when the *teacher* returns *continue* and a counterexample *cex* that the *learner* can use to learn a better candidate Boolean function after analyzing, the algorithm continues running to create new *conjecture* functions and submitting new conjectures to the *teacher* (steps 6, 7 and then 2). This process repeats until the *teacher* returns either *yes* or *no* and a counterexample *cex* that witnesses the fact that $M_0||M_1 \not\models \pi$. Thus, these are the same as the above two described cases in which the algorithm returns *yes* or *no and cex*. □

Lemma 3 *(Termination). LWAG algorithm terminates in a finite number of membership queries and equivalence queries.*

Proof. CBAG algorithm terminates within a polynomial number of queries in $|\iota_1(X_1)|_{DNF}$, $|\iota_1(X_1)|_{CNF}$, $|\tau_1(X_1, X_1')|_{DNF}$, $|\tau_1(X_1, X_1')|_{CNF}$, and $|X_1|$ [13]. In LWAG algorithm, if the *teacher* returns *yes* (step 2), then the algorithm stops. If the *teacher* returns a *no* result and a counterexample *cex* with which the *learner* cannot learn another Boolean function for a new *conjecture* (i.e., a check for "violation?" returns *yes*) (a real counterexample), the *learner* will update one *question* result to *no* (if any exist) (step 3) and continues trying to learn another *conjecture* (steps 4 and 5). If the query continues to return a *no* result and a real counterexample, eventually, there will be no more *question* responses in the membership query result list because the number of *question* results in the list is finite. At that point, the algorithm terminates thanks to the correctness of CBAG algorithm presented in Section 3. Otherwise, when the *teacher* returns *continue* and a counterexample *cex* with which the *learner* can learn a new Boolean function for a new *conjecture* after analyzing (i.e., a check for "violation?" returns *no*), the algorithm continues running, creating new *conjecture* functions and submitting new conjectures to the *teacher* (steps 6 and 7). This process repeats until the *teacher* returns either *yes* or a *no* result and a real counterexample *cex*. The number of steps needed to create *conjecture* functions is also finite because the CDNF algorithm can learn any Boolean function in a finite number of steps [10]. Therefore, the process for generating a new *conjecture* will

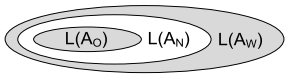

Fig. 3. Relationships among $L(A_O)$, $L(A_N)$, and $L(A_W)$.

terminate in a finite number of steps. Eventually, this case becomes one of the above two described cases and the algorithm will terminate. □

Lemma 4 *(Complexity). In the worst case, LWAG algorithm terminates in* $n^2.Size_{DNF}.Size_{CNF}$ *membership queries and* $\sum_{i=1}^{Size_{DNF}.Size_{CNF}}(Size_{DNF}.Size_{CNF}-i+1)$ *equivalence queries to learn the corresponding* ι_A *or* τ_A *function.*

Proof. To implement LWAG algorithm, we need to implement two *learner* instances using the CDNF algorithm [10] to learn ι_A and τ_A. CBAG algorithm requires $Size_{DNF}.Size_{CNF}$ equivalence queries and $n^2.Size_{DNF}.Size_{CNF}$ membership queries to learn one Boolean function, where n is the basis size [10]. In LWAG algorithm, to learn the ι_A or τ_A function (because the *learner* stores previous membership query results) the total number of membership queries is the same as CBAG algorithm in the worst case, which is $n^2.Size_{DNF}.Size_{CNF}$. With each counterexample *cex* returned from an equivalence query, the algorithm creates a backtracking point by backing up the status of the corresponding CDNF algorithm (step 6). Therefore, we will have $Size_{DNF}.Size_{CNF}-1$ backtracking points. This is because the last answer from the *teacher* will be a real counterexample (a counterexample from which the *learner* cannot learn another Boolean function to create a new *conjecture*), and the *learner* will not create a backtracking point for this answer. For each backtracking point, the *learner* needs to turn some *question* membership query answers to *no* (step 4) and then send the *teacher* a new equivalence query (steps 5 then 2). In the worst case, with the i^{th} backtracking point (where $1 \le i < Size_{DNF}.Size_{CNF}$), the algorithm will require $Size_{DNF}.Size_{CNF}-i+1$ equivalence queries. This is because we need $Size_{DNF}.Size_{CNF}$ equivalence queries to learn the function at step i^{th} in total – but we already asked $i-1$ equivalence queries before each step. Therefore, in total, and in the worst case, we will need $\sum_{i=1}^{Size_{DNF}.Size_{CNF}}(Size_{DNF}.Size_{CNF}-i+1)$ equivalence queries. This complexity is clearly greater than the complexity of CBAG algorithm. □

From Lemma 4, the worst-case complexity of LWAG algorithm is greater than that of CBAG algorithm. In regards to the average-case complexity, the experiment results presented in Section 6 show that LWAG algorithm takes longer to generate assumptions than CBAG algorithm. However, in general, there are some scenarios where LWAG algorithm is faster than CBAG algorithm. In such cases, the best-case complexity of LWAG algorithm is less than that of CBAG algorithm.

Lemma 5 *(Language relationship). Let* A_O, A_N, *and* A_W *be assumptions generated by CBAG algorithm, LWAG algorithm, and the weakest assumption, respectively. The relationship among* $L(A_O)$, $L(A_N)$, *and* $L(A_W)$ *is as follows:* $L(A_O) \subseteq L(A_N) \subseteq L(A_W)$.

Proof. We consider the following cases to prove the correctness of Lemma 5. Because A_W is the weakest assumption, we always have $L(A_N) \subseteq L(A_W)$. In the case where no *question* result exists when the *teacher* answers *yes* to the equivalence query, the final assumption is the same as the assumption generated by CBAG algorithm. Therefore, in this case, $L(A_N)$ is equal to $L(A_O)$. The last case is the case where some *question* results exist when the *teacher* answers *yes* to the equivalence query. From Table 2, this means the cases in which $X = F$ but $Y = T$ is integrated into the final accepted Boolean functions of ι_A and τ_A. To prove $L(A_O) \subseteq L(A_N)$, we prove that $\forall \alpha \in L(A_O)$, we also have $\alpha \in L(A_N)$, where $\alpha = \mu^0\mu^1...\mu^t \in L(A_O)$. According to the *Trace* definition, we have $\iota_{A_O}[\mu^0] = T$ and $\tau_{A_O}[\mu^i, \mu^{i+1}] = T$ for $0 \le i < t$. From OMQ algorithm, we have $\iota_1[\mu^0] = T$ and $\tau_1[\mu^i, \mu^{i+1}] = T$ for $0 \le i < t$. Moreover, thanks to IMQ algorithm, we have $\iota_{A_N}[\mu^0] = T$ and $\tau_{A_N}[\mu^i, \mu^{i+1}] = T$ for $0 \le i < t$. This means that $\forall \alpha : \alpha \in L(A_N)$. That is $L(A_O) \subseteq L(A_N)$. Fig. 3 illustrates the relationships among $L(A_O)$, $L(A_N)$, and $L(A_W)$. □

Lemma 6 *(Local weakest assumption). Assume that LWAG algorithm does not return the assumption immediately after obtaining the first satisfied assumption; instead, it continues running to find all possible assumptions until all of the* `question` *results have been changed into no results in the list.*

Let $\mathfrak{A}$ *be the above set of assumptions and A be the first generated assumption. A is the local* `weakest assumption` *in* $\mathfrak{A}$ *(Remark 1).*

Proof. Let the first-found assumption be A_{NW} and an arbitrary assumption found by LWAG algorithm after A_{NW} be A_N. Let the list of membership query results corresponding to A_{NW} and A_N be $List_{NW}$ and $List_N$, respectively. We prove that $L(A_N) \subseteq L(A_{NW})$. To do this, we prove that $\forall \alpha \in L(A_N)$; we also have $\alpha \in L(A_{NW})$, where $\alpha = \mu^0\mu^1...\mu^t \in L(A_N)$. According to the *Trace* definition, we have $\iota_{A_N}[\mu^0] = T$ (*true*) and $\tau_{A_N}[\mu^i, \mu^{i+1}] = T$ for $0 \le i < t$. In LWAG algorithm, when it reaches a satisfied assumption, the algorithm already considers all the existing *question* results in the list as *yes* (steps 5 and 7). Because A_{NW} is the first found assumption, assume that A_N is found after n steps of changing *question* result to *no*, where $n > 0$. We can easily see that all the *yes* items in $List_N$ also exist in $List_{NW}$. Therefore, we have $\iota_{A_{NW}}[\mu^0] = T$ and $\tau_{A_{NW}}[\mu^i, \mu^{i+1}] = T$ for $0 \le i < t$. Consequently, we obtain $\alpha \in L(A_{NW})$. The relationship between $List_{NW}$ and $List_N$ is

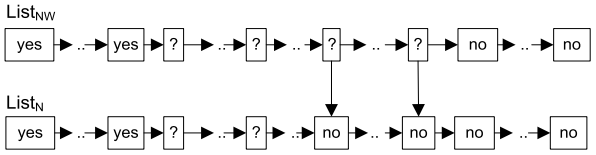

Fig. 4. The relationship between $List_{NW}$ and $List_N$.

shown in Fig. 4 (where the *question* results are represented as *question mark* ("?") symbols). We can see that after some steps of changing a *question* result to *no* (step 4) in $List_{NW}$, we will have $List_N$. □

Thus far, we have presented LWAG algorithm that uses an improved technique to answer membership queries. This algorithm generates weaker assumptions than those generated by CBAG algorithm. Although LWAG algorithm has a greater complexity than does CBAG algorithm, the generated assumptions can reduce the number of times assumptions must be regenerated when rechecking a modified system in the context of software evolution. In the long run, where most of the effort is spent on software maintenance, this will significantly reduce the verification cost for rechecking evolving systems, especially large-scale systems. To reduce the verification cost of evolving systems, Section 5 shows an effective framework for using these weaker assumptions to recheck evolving CBSs.

5. A framework for modular verification of evolving CBS

In practice, when software verification cost increases daily because of software evolution which can happen all time during software life cycle, more reusable assumptions, such as weak assumptions, play an important role in reducing verification cost by being used in the framework presented in this section. The empirical results shown in Section 6 clearly indicates the effectiveness of using weak assumptions when rechecking evolved software.

Consider a CBS M that contains two components M_0 and M_1. Assume that M_0 is a type of static framework component that remains unchanged during the software life cycle. M_1 is a business/extension component that is supposed to change during software evolution. Let A be an assumption under which M satisfies a predicate π. When software is modified, there are several types of change that we must consider, as follows.

1. When some existing behaviors of M_1 are removed, it becomes M_1' with $L(M_1') \subseteq L(M_1)$. Then, we already have $L(M_1') \subseteq L(A)$ because $L(M_1) \subseteq L(A)$. Therefore, the assumption will not need to be regenerated.
2. When updating some existing behaviors of M_1, it becomes M_1' with $L(M_1') \subseteq L(M_1)$. In this case, we already have $L(M_1') \subseteq L(A)$ because $L(M_1) \subseteq L(A)$. Therefore, the assumption will not need to be regenerated.
3. When updating some existing behaviors of M_1, it becomes M_1' with $L(M_1') \nsubseteq L(M_1)$ and $L(M_1') \subseteq L(A)$. This assumption will not need to be regenerated.
4. When updating some existing behaviors of M_1, it becomes M_1' with $L(M_1') \nsubseteq L(M_1)$ and $L(M_1') \nsubseteq L(A)$; therefore, the assumption will need to be regenerated.
5. When adding some new behaviors to M_1, it becomes M_1' with $L(M_1') \subseteq L(M_1)$; however, we already have $L(M_1') \subseteq L(A)$ because $L(M_1) \subseteq L(A)$. Therefore, the assumption will not need to be regenerated.
6. When adding some new behaviors to M_1, it becomes M_1' with $L(M_1') \nsubseteq L(M_1)$ and $L(M_1') \subseteq L(A)$. The assumption will not need to be regenerated.
7. When adding some new behaviors to M_1, it becomes M_1' with $L(M_1') \nsubseteq L(M_1)$ and $L(M_1') \nsubseteq L(A)$. Therefore, this assumption will need to be regenerated.

From the above types of change, we can see that numbers 4 and 7 require A to be regenerated because $L(M_1') \nsubseteq L(A)$. Therefore, the greater $L(A)$ is, the greater $L(M_1')$ can be such that $L(M_1') \subseteq L(A)$ (i.e., more behaviors can be added to M_1 such that $L(M_1') \subseteq L(A)$). Consequently, the greater $L(A)$ is (i.e., the weaker A is), the more cost of software verification can be reduced because the number of times that A can be reused is increased. Consequently, weak assumptions play a key role in reducing the software verification cost in the context of software evolution.

As shown in Section 4, the assumption generated by LWAG algorithm A_N is weaker than the assumption A_{Org} generated by CBAG algorithm. Therefore, in the context of software evolution, the assumption A_N can be used for modular verification in modified CBSs to reduce the verification cost as shown in Fig. 5.

As proposed in the framework by Hung et al., the previous assumption A_{Org} can be reused when verifying CBS with modified M_1 [33]. This situation is much better than the one in which we need to restart the assumption learning process all over again from the beginning. Using LWAG algorithm presented in Section 4, the generated assumption A_N has $L(A_{Org}) \subseteq L(A_N)$. As a result, by using A_N as the starting point for verifying the modified CBS, we can reduce the number of times that assumptions must be regenerated from A_{Org} to A_N as shown in Fig. 5.

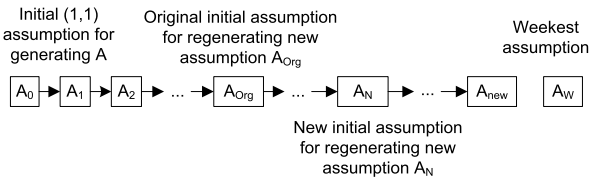

Fig. 5. Reusing assumptions generated by LWAG algorithm for evolving CBS.

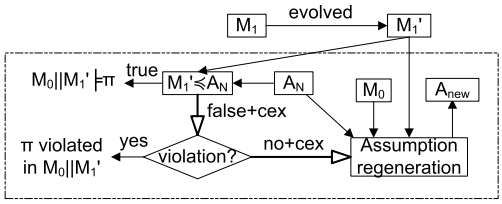

Fig. 6. The algorithm to regenerate assumption for evolving CBS.

5.1. The proposed framework

This section proposes a framework for verifying CBSs in the context of component change. This framework was developed based on the framework proposed by Hung et al. [33,32]. The proposed framework is shown in Fig. 6 and has the following steps.

Let $M = M_0||M_1$ be a component-based software, π be a predefined predicate, and A_N be the assumption generated by LWAG algorithm.

1. Here, Model M_1 of a component-based software $M_0||M_1$ evolves during the software life cycle. Assume that M_1' is the modified model of M_1.
2. The previous assumption A_N is used as the starting assumption for the reverification process. A_N is checked to see if $M_1' \preceq A_N$. If $M_1' \preceq A_N$, then $M_0||M_1' \models \pi$ because we already have $M_0||A_N \models \pi$. If $M_1' \not\preceq A_N$, then this step returns *false* and a counterexample, *cex*.
3. The returned *cex* is analyzed to see if the modified system ($M_0||M_1'$) truly violates the property. If it does, then we have $M_0||M_1' \not\models \pi$. Otherwise, we will need to generate a new assumption.
4. The new assumption A_{new} is generated using A_N as a starting candidate, the counterexample *cex*, and M_0. To avoid learning assumptions that had been sent to the *teacher*, the *learner* needs to store required information for checking candidate and membership queries duplication. This is a simple task and is not mentioned in both LWAG algorithm and the framework shown in Fig. 6.

Although the framework in Fig. 6 shows the simple case in which the CBS is composed of only two component models M_0 and M_1, we can generalize it to larger systems containing n-component models, $M_0, M_1, ..., M_n$, where $n \geq 2$. Nevertheless, the framework for a larger system consists of similar steps as described above because we focus only on the modified component models.

In 2016, He et al. proposed a fast assumption generation method [27] by learning the subpredicates of the assumption to be generated simultaneously. However, even when using the assumptions generated by He et al.'s method, the framework proposed by Hung et al. [33,32] still needs to regenerate the assumption every time even a small change occurs in the component model. Thus, over the full software life cycle, even when using the more effective assumption generation method, the regeneration cost may still be very high.

5.2. An example

This section shows an example of generating an assumption using LWAG algorithm and the framework shown in Section 5.1 for verifying an evolving system. Consider a system $M = M_0||M_1$, where $M_i = \langle X_i, \iota_i(X_i), \tau_i(X_i, X_i')\rangle$ is a transition system for i = 0,1, and π is a state predicate over $X_0 \cup X_1$ as shown below. In this example, we use the notation "$CDNF_\iota$" for the CDNF algorithm instance for learning ι_A and "$CDNF_\tau$" for the CDNF algorithm instance for learning τ_A. We also use (ι_A, τ_A) to represent a candidate assumption without loss of generality.

$X_{M_0} = \{1, 2\}$,
$X'_{M_0} = \{3, 4\}$,
$\iota_{M_0} = (-1 \wedge -2)$,
$\tau_{M_0} = (-1 \wedge -2 \wedge -3 \wedge 4)|(-1 \wedge 2 \wedge 3 \wedge -4)|(1 \wedge -2 \wedge 3 \wedge 4)|(1 \wedge 2 \wedge -3 \wedge -4)$, and
$X_{M_1} = \{5, 6\}$,

Table 3
Generation of the first assumption using LWAG algorithm.

Step	Action	Result
Step 1	Set $\iota_A = T$, $\tau_A = T$	$\iota_A = T$, $\tau_A = T$
Step 2	Ask Equivalence Query (EQ) for ($\iota_A = T$, $\tau_A = T$)	Return 0100 to $CDNF_\tau$
Step 6	Backup status	-
Step 7	Try to learn new τ_A	$\iota_A = T$, $\tau_A = (6)$
Step 2	Ask EQ for ($\iota_A = T$, $\tau_A = (6)$)	Return 0001 to $CDNF_\tau$
Step 6	Backup status	-
Step 7	Try to learn new τ_A	$\iota_A = T$, $\tau_A = (6)\|(8)$
Step 2	Ask EQ for ($\iota_A = T$, $\tau_A = (6)\|(8)$)	Return 0101 to $CDNF_\tau$
Step 6	Backup status	-
Step 7	Try to learn new τ_A	$\iota_A = T$, $\tau_A = (6)\|(8)\|(6 \wedge 8)$
Step 2	Ask EQ for ($\iota_A = T$, $\tau_A = (6)\|(8)\|(6 \wedge 8)$)	Return 0011 to $CDNF_\tau$
Step 6	Backup status	-
Step 7	Try to learn new τ_A. While learning, it asked membership query for 0010 and the answer is *question*. This is first considered as T (True).	$\iota_A = T$, $\tau_A = (6)\|(8)\|(6 \wedge 8)\|(7)$
Step 2	Ask EQ for ($\iota_A = T$, $\tau_A = (6)\|(8)\|(6 \wedge 8)\|(7)$)	Return 10 to $CDNF_\iota$
Step 6	Backup status	-
Step 7	Try to learn new ι_A	$\iota_A = (5)$, $\tau_A = (6)\|(8)\|(6 \wedge 8)\|(7)$
Step 2	Ask EQ for ($\iota_A = (5)$, $\tau_A = (6)\|(8)\|(6 \wedge 8)\|(7)$)	Return 00 to $CDNF_\iota$
Step 6	Backup status:	-
Step 7	Try to learn new ι_A. While learning, it asked membership query for 11 and the answer is *question*. This is first considered as T (True).	$\iota_A = (5)\|(-6)$, $\tau_A = (6)\|(8)\|(6 \wedge 8)\|(7)$
Step 2	Ask EQ for ($\iota_A = (5)\|(-6)$, $\tau_A = (6)\|(8)\|(6 \wedge 8)\|(7)$)	Return *yes*

$X'_{M_1} = \{7, 8\}$,
$\iota_{M_1} = (-5 \wedge -6)$,
$\tau_{M_1} = (-5 \wedge -6 \wedge -7 \wedge 8)|(-5 \wedge 6 \wedge 7 \wedge -8)|(5 \wedge -6 \wedge 7 \wedge 8)|(5 \wedge 6 \wedge -7 \wedge -8)$,
$\pi = (-1 \wedge -2 \wedge -5 \wedge -6)|(-1 \wedge 2 \wedge -5 \wedge 6)|(1 \wedge -2 \wedge 5 \wedge -6)|(1 \wedge 2 \wedge 5 \wedge -6)|(1 \wedge 2 \wedge 5 \wedge 6)|(1 \wedge -2 \wedge 5 \wedge 6)|(1 \wedge 2 \wedge -5 \wedge -6)|(1 \wedge 2 \wedge -5 \wedge 6)$.

5.2.1. Generating the first assumption

The first step of the proposed framework is to generate an assumption A_N that satisfies Definition 6. Table 3 shows how LWAG algorithm generates the assumption A_N. From the result of LWAG algorithm shown in Table 3, the generated assumption A_N is as follows:
$X_{A_N} = X_{M_1} = \{5, 6\}$,
$X'_{A_N} = X'_{M_1} = \{7, 8\}$,
$\iota_{A_N} = (5)|(-6)$, and
$\tau_{A_N} = (6)|(8)|(6 \wedge 8)|(7)$.
In the meantime, the result of CBAG algorithm forms a stronger assumption, A_O, as follows:
$X_{A_O} = X_{M_1} = \{5, 6\}$,
$X'_{A_O} = X'_{M_1} = \{7, 8\}$,
$\iota_{A_O} = (-6)$, and
$\tau_{A_O} = (6)|(8)|(6 \wedge 8)|(7 \wedge 8)$.

5.2.2. Verifying evolving systems

Consider the case in which M_1 is modified by adding the predicate $(-5 \wedge -6 \wedge 7 \wedge -8)$ to τ_{M_1}, causing M_1 to become the component M'_1 as follows:
$X_{M'_1} = \{5, 6\}$,
$X'_{M'_1} = \{7, 8\}$,

Table 4
Generating an assumption for an evolving system using LWAG algorithm and the proposed framework.

Step	Action	Result
Step 2	Ask EQ for $((5)\|(-6), (6)\|(8)\|(6 \wedge 8)\|(7))$	Return 1000 to $CDNF_\tau$
Step 6	Make a copy of status	-
Step 7	Try to learn new τ_A	$\iota_A = (5)\|(6), \tau_A = (6)\|(8)\|(6 \wedge 8)\|(7)\|(5)$
Step 2	Ask EQ for $((5)\|(-6), (6)\|(8)\|(6 \wedge 8)\|(7)\|(5))$	Return *yes*

Table 5
Assumption generation methods comparison.

Test cases	Common			CBAG algorithm						LWAG algorithm					
	B	$\|X_0\|$	$\|X_1\|$	EQ	I	T	Time (ms)	Mem (B)	$\|L(A_O)\|$	EQ	I	T	Time (ms)	Mem (B)	$\|L(A_N)\|$
TC1_0	4	2	2	8	2	1	3,093	1,371,782	70	11	1	1	4,032	1,401,310	177
TC2_0	4	2	2	8	2	2	1,475	1,472,883	94	10	1	1	1,737	1,458,893	182
TC3_0	4	2	2	6	2	2	1,665	1,284,175	30	10	1	1	3,007	1,405,921	163
TC4_0	4	2	2	6	2	2	1,649	1,359,926	30	10	1	1	2,808	1,441,094	163
TC5_0	4	3	3	10	5	5	31,486	1,711,054	340	18	3	3	117,112	1,704,387	3,343
TC6_0	4	4	4	18	9	10	1,457,649	5,511,856	4,680	34	7	7	6,697,195	5,760,382	59,455

$\iota_{M'_1} = (-5 \wedge -6)$,
$\tau_{M'_1} = (-5 \wedge -6 \wedge -7 \wedge 8)|(-5 \wedge 6 \wedge 7 \wedge -8)|(5 \wedge -6 \wedge 7 \wedge 8)|(5 \wedge 6 \wedge -7 \wedge -8)|(-5 \wedge -6 \wedge 7 \wedge -8)$.

We can easily see that $M'_1 \preceq A_N$ but $M'_1 \not\preceq A_O$. Therefore, there is no need to regenerate the assumption when A_N is used as a starting assumption in the proposed framework in Section 5.1. However, if A_O were to be used as a starting assumption in the framework, we would need to generate another assumption.

In the case where M_1 is modified by adding the predicate $(5 \wedge -6 \wedge -7 \wedge -8)$ to τ_{M_1}, M_1 becomes the component M''_1 as follows:
$X_{M''_1} = \{5, 6\}$,
$X'_{M''_1} = \{7, 8\}$,
$\iota_{M''_1} = (-5 \wedge -6)$,
$\tau_{M''_1} = (-5 \wedge -6 \wedge -7 \wedge 8)|(-5 \wedge 6 \wedge 7 \wedge -8)|(5 \wedge -6 \wedge 7 \wedge 8)|(5 \wedge 6 \wedge -7 \wedge -8)|(5 \wedge -6 \wedge -7 \wedge -8)$.

We can see that $M''_1 \not\preceq A_N$ and $M''_1 \not\preceq A_O$. Therefore, regardless of whether A_O or A_N is used as the starting assumption in the framework, we will need to generate new assumptions. Table 4 shows how LWAG algorithm and the framework in Section 5 are applied to verify the evolving system for the one mentioned in the beginning of Section 5.2 when M_1 evolves to be M''_1.

6. Experiments

To evaluate the effectiveness of LWAG algorithm, experiments are performed to highlight two key points: (i) a comparison between CBAG algorithm and LWAG algorithm and their corresponding generated assumptions; and (ii) a comparison of the framework in Section 5.1 between the cases using the assumptions generated by CBAG algorithm and LWAG algorithm after the software has been modified. Algorithms presented in Section 3 and Section 4 are implemented in *C#.NET* and Microsoft Visual Studio Community 2017. The verification tool is called AGVerifier; and is available from http://www.tranhoangviet.name.vn/p/agverifier.html. AGVerifier is based on the CELL framework [36] and includes ready-to-use test cases. Those test cases and the evolved ones are described in Section 6.1 and 6.2. We used a bounded model–checking approach to conduct the experiments in which the bound B was selected so that the longest traces in both components of those test cases can cover their transitions between all their states. The experiments are performed on a machine with following specifications: Microsoft Windows 10 Home edition operating system, Intel Core i5-5200U 2.2 Ghz CPU, 8.00 GBs RAM memory. To present reliable experimental results, each test case are performed 10 times and the average results are reported in Table 5 and Table 6.

6.1. Assumption generation algorithms comparison

To compare CBAG algorithm with LWAG algorithm with the corresponding generated assumptions, we used the same test data for both and compared the same key indicators: the number of equivalence queries for assumption candidates, the number of membership queries for the initial and transition functions, the time needed to generate assumptions, memory usage, and the size of the languages of the generated assumptions. We performed experiments with the following systems.

- ***Candy Packaging Line Controller*** The candy packaging line controller (denoted by TC1) is a system that controls the process of candy packaging. The controller is a part of one of our national projects. The design of this controller is based on the system called Simple Communication Channel from Cobleigh et al.'s paper [19]. The controller consists of

Table 6
The generated assumptions comparison in software evolution context.

Test cases	CBAG algorithm						LWAG algorithm					
	EQ	I	T	Time(ms)	Mem (B)	$\|L(A_{RO})\|$	EQ	I	T	Time(ms)	Mem (B)	$\|L(A_{RN})\|$
TC1_0.1	-	-	-	-	-	-	-	-	-	-	-	-
TC1_0.2	3	0	1	620	1,526,210	108	-	-	-	-	-	-
TC1_0.3	2	0	0	376	1,546,480	108	2	0	0	680	1520334	215
TC1_0.4	3	0	1	610	1,403,568	108	-	-	-	-	-	-
TC1_0.4.1	3	0	1	629	1,270,924	108	-	-	-	-	-	-
TC2_0.1	2	0	0	211	1,439,318	115	-	-	-	-	-	-
TC2_0.1.1	2	0	0	215	1,565,050	115	-	-	-	-	-	-
TC2_0.1.2	2	0	0	219	1,332,717	115	-	-	-	-	-	-
TC2_0.1.2.1	2	0	0	228	1,219,572	115	-	-	-	-	-	-
TC3_0.1	5	0	3	981	1,533,133	96	-	-	-	-	-	-
TC3_0.1.1	5	0	3	992	1,445,293	96	-	-	-	-	-	-
TC3_0.1.1.1	6	0	6	1,250	1,439,177	132	5	0	3	1,775	1,525,369	208
TC3_0.1.1.1.1	5	0	3	998	1,597,769	96	-	-	-	-	-	-
TC4_0.1	2	0	2	282	1,554,553	62	-	-	-	-	-	-
TC4_0.1.1	2	0	2	278	1,701,371	62	-	-	-	-	-	-
TC4_0.1.1.1	2	0	2	267	1,469,968	62	-	-	-	-	-	-
TC4_0.1.1.1.1	2	0	2	259	1,633,011	62	-	-	-	-	-	-
TC4_0.1.1.1.1.1	2	0	2	294	1,662,419	62	-	-	-	-	-	-
TC5_0.1	-	-	-	-	-	-	-	-	-	-	-	-
TC5_0.2	11	0	25	29,119	1,714,294	1,276	-	-	-	-	-	-
TC5_0.2.1	10	0	19	25,036	1,699,504	1,276	-	-	-	-	-	-
TC5_0.2.1.1	10	0	19	25,537	1,702,160	1,276	-	-	-	-	-	-
TC5_0.2.1.1.1	9	0	18	21,508	1,713,168	1,276	-	-	-	-	-	-
TC6_0.1	21	0	50	1,071,687	5,705,728	18,792	-	-	-	-	-	-
TC6_0.1.1	21	0	49	1,096,787	5,735,795	18,792	-	-	-	-	-	-
TC6_0.1.1.1	20	0	43	1,080,497	6,008,460	18,792	-	-	-	-	-	-
TC6_0.1.1.1.1	20	0	43	1,051,336	5,292,621	18,792	-	-	-	-	-	-
TC6_0.1.1.1.1.1	19	0	44	1,061,336	5,778,549	18,792	-	-	-	-	-	-

two components M_0 and M_1. M_0 has four states: *in*: candy is poured into a package; *process*: the package with candy is weighed to see if its weigh is correct or not; *send*: the package with candy is sent to the section where it is soldered; *ack*: the system finished soldering the candy package and it is ready to process other candy packages. M_1 has four states: *process*: the candy package is weighed to see if its weigh is correct or not; *send*: the candy package is sent to the section where it is soldered; *out*: candy package information is displayed on user's monitor; *ack*: the system finished displaying information and it is ready to process other candy packages. We checked the property that all following restrictions need to be satisfied: The system states must be in the following order: in $\rightarrow$ process $\rightarrow$ send / out $\rightarrow$ ack; after a package is soldered or its information is displayed, candy can be poured into the next package; when the system finished packaging the current package, candy can be poured into the next package or the next package can be weighed or the next package can be soldered or the next package information can be displayed; the system can have some *spare* time before candy can be poured into the next package. For this candy packaging line controller, we used two Boolean variables to encode the states of M_0 and M_1 (i.e., $|X_0| = 2$ and $|X_1| = 2$).

- ***Waste sorting line controller*** The waste sorting line controller (denoted by TC2) is a system that controls the process of sorting and processing waste into two categories of organic and inorganic. The controller is another part of the national project mentioned above. The controller has two components M_0 and M_1. M_0 has three states: *in*: an amount of waste is put into the system and sorted; *organic waste*: waste is recognized to be organic; *inorganic waste*: waste is recognized to be inorganic. M_1 also has three states: *sorted waste*: sorted waste (either organic or inorganic) is received and prepared to be processed; *process*: depends on the type of waste, it is passed to the recycling phase (inorganic waste) or used to produce chemical fertilizers (organic waste); *ack*: waste processing is finished and the component is ready to receive another amount of waste to work on. We checked the property that all following restrictions need to be satisfied: an amount of waste can only be recognized as organic or inorganic after it is received and sorted; the amount of waste can only be processed after it is sorted into either organic or inorganic; only after an amount of waste is sorted and processed, the system is ready to receive another amount of waste to work on. For this waste sorting line controller, we used two Boolean variables to encode the states of M_0 and M_1 (i.e., $|X_0| = 2$ and $|X_1| = 2$).
- ***Master / slave system*** The master / slave system (denoted by TC3) is one of the example system presented in Magee and Kramer's book [43]. This is a typical system where a master thread creates a slave thread to perform some tasks such as I/O and continues its work. Later, the master synchronizes with the slave to get the result. The system consists of two components M_0 and M_1. M_0 (master) has four states: *slave.start*: the master created a new slave thread; *rotate1*: the master does its own work; *slave.join*: the master synchronizes with the slave to get the result; *rotate2*: the master does its own work and gets ready to create a new slave. M_1 (slave) has three states: *slave.start*: the slave is started; *slave.rotate*: the slave does its own work; *slave.join*: the master synchronized with the slave to get the result and then, the slave is ready to be started again. We checked the property that all following restrictions need to be satisfied: after

the master starts a slave, it can wait for some time before doing its own work (rotate1); later, it can synchronize with the slave to get the result; the master then can continue its own work (rotate2) and is ready to start a new slave. For this master / slave system, we used two Boolean variables to encode the states of M_0 and M_1 (i.e., $|X_0| = 2$ and $|X_1| = 2$).

- ***Simple communication channel*** The simple communication channel (denoted by TC4) is a common example system used in Cobleigh et al.'s paper [19] that controls the process of receiving and sending message / data in a communication channel. The channel has two components M_0 and M_1. M_0 has three states: *in*: the channel received a message; *send*: the message is sent to another channel; *ack*: the channel is ready to receive another message; M_1 has three states: *send*: the message is sent to another channel; *out*: the channel provides feedback to the monitor thread; *ack*: the channel is ready to send another message. We tested the channel with a more complex safety property that all following restrictions need to be satisfied: the channel can only send a message after it receives the message; the channel can provide feedback to the monitor thread after it receives and sends a message; the channel is ready to receive another message when it finished sending the previous message and providing feedback to the monitor thread. For this simple communication channel, we used two Boolean variables to encode the states of M_0 and M_1 (i.e., $|X_0| = 2$ and $|X_1| = 2$).
- ***Simple communication channel - variant 1*** The simple communication channel - variant 1 (denoted by TC5) is the same as the simple communication channel used in Cobleigh et al.'s paper [19]. However, we used three Boolean variables to encode the states of M_0 and M_1 (i.e., $|X_0| = 3$ and $|X_1| = 3$). This is to check the affection of the number of Boolean variables inside a transition system to the verification process. This test case can also show that our proposed verification method can be used to check software systems represented by Label Transition System (LTS) whose maximum sizes can be $size\ of\ component\ 0 \times size\ of\ component\ 1 \times (size\ of\ property + 1) = |C_0| \times |C_1| \times |p_{err}| = 2^3 \times 2^3 \times (2^3 + 1) = 576$ [35]. An LTS C 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 C; τ represents the unobservable action of C to its environment; $\delta \subseteq Q \times \Sigma \cup \{\tau\} \times Q$ is a transition relation; and $q_0 \in Q$ is the initial state. When we check whether an LTS C satisfies a required safety property p, an *error LTS*, denoted by p_{err}, is created which traps possible violations with the Π state (i.e., the error state). p_{err} of a property $p = \langle Q, \Sigma_p, \delta, q_0 \rangle$ is $\langle Q \cup \{\Pi\}, \Sigma_p, \delta', q_0 \rangle$, where $\delta' = \delta \cup \{(q, a, \Pi) \mid a \in \Sigma_p \text{ and } \nexists q' \in Q : (q, a, q') \in \delta\}$.
- ***Simple communication channel - variant 2*** The simple communication channel - variant 2 (denoted by TC6) is the same as the simple communication channel used in Cobleigh et al.'s paper [19]. However, we used four Boolean variables to encode the states of M_0 and M_1 (i.e., $|X_0| = 4$ and $|X_1| = 4$). This is to check the affection of the number of Boolean variables inside a transition system to the verification process. This test case can also show that our proposed verification method can be used to check software systems represented by Label Transition System (LTS) whose maximum sizes can be $size\ of\ component\ 0 \times size\ of\ component\ 1 \times (size\ of\ property + 1) = |C_0| \times |C_1| \times |p_{err}| = 2^4 \times 2^4 \times (2^4 + 1) = 4352$ [35].

Table 5 shows the results of the experiments. The columns "Test cases", "B", $|X_0|$, and $|X_1|$ contain the test case short name in which "*_0*" means version 0, the trace length's bound number, and the sizes of the Boolean variable sets of M_0 and M_1, respectively. For example, in Table 5, "TC1_0" in line 1 indicates the experimental results for the version 0 of TC1 (Candy Packaging Line Controller), and so on. For simplicity without loosing the generality of the proposed method, the bound B was selected so that the longest traces in both components of those test cases can cover their transitions between all their states. The columns "EQ", "I", "T", "Time(ms)", and "Mem(B)" are the number of equivalence queries, the initial and transition membership queries, the time (in milliseconds), and the memory (in bytes) required to generate assumptions, respectively. $|L(A_O)|$ and $|L(A_N)|$ are the sizes of the corresponding languages of the assumptions generated by CBAG algorithm and the improved algorithm, respectively. We calculated $|L(A_O)|$ and $|L(A_N)|$ by counting the number of traces in $|L(A_O)|$ and $|L(A_N)|$ with the bound shown in the "B" column of Table 5.

From the experimental results shown in Table 5, we can see that the number of queries for assumption candidates, initial function candidates, and transition function candidates of the two algorithms are different; the assumptions generated by LWAG algorithm are weaker than the assumptions generated by CBAG algorithm. However, LWAG algorithm takes longer to generate the assumptions because it requires more processing (i.e., it must process more assumption candidate queries). However, the amount of memory used by the two algorithms is similar.

6.2. The effectiveness of the generated assumptions in software evolution context

To compare the effectiveness of the assumptions generated by LWAG algorithm compared to those generated by CBAG algorithm in a software evolution context, we implemented the proposed framework described in Section 5 for verifying modified systems for the test cases shown in Table 5. When performing the experiments, we also measured the same key indicators as presented in Section 6.1. However, we focus on different key information from the experimental results – that is, whether a previous assumption can be reused when verifying modified systems (i.e., whether we can avoid regenerating assumptions unnecessarily when verifying modified systems).

Table 6 shows the experimental results. "*_x.n*" indicates that "*version x.n was evolved from version x*". For example, "TC1_0.1" and "TC1_0.4.1" were evolved from versions "TC1_0" and "TC1_0.4", respectively. In this table, test cases are modified systems of those tested in Section 6.1, in all of which M_0 was kept unchanged but M_1 was modified from M_1 of the corresponding previous version. Details of the test cases in Table 6 are described below.

"TC1_0.1" – "TC1_0.4.1" are modified systems from "TC1_0", in all of which M_1 was modified from M_1 of the previous version by adding some behaviors as follows. In TC1_0.1, the system can only be ready for processing the next package when the current package was weighed, soldered, and the system finished displaying information on user's monitor. In TC1_0.2, the system needs to satisfy the following requirements: after a candy package is weighed or soldered, the information is displayed on user's monitor and the system is ready to process another candy package; the information can be displayed on user's monitor for some time before processing another candy package; the system can be ready for some time before it receives another package of candy. In TC1_0.3, when the system finished displaying information on user's monitor, it can receive another candy package. In TC1_0.4, after a package of candy is weighed, the information about the package can be displayed on user's monitor. In TC1_0.4.1, the information can be displayed on user's monitor for some time before the system is ready to process another candy package (e.g., it is waiting for command from user).

Similar to "TC1", "TC2_0.1" – "TC2_0.1.2.1" are modified systems from "TC2_0" by adding some behaviors as follows. In TC2_0.1, when the system finished processing a certain amount of waste, it is ready to receive another sorted amount of waste to process. In TC2_0.1.1, the process of the current amount of waste can take some time before the system can be ready for receiving another amount of sorted waste to process. In TC2_0.1.2, after processing a certain amount of waste, the information is displayed on the screen of the user. Then, the system is ready to receive another amount of sorted waste to process. In TC2_0.1.2.1, the process of the current amount of waste can take some time before the system can be ready for receiving another amount of sorted waste to process.

"TC3_0.1" – "TC3_0.1.1.1.1" are modified systems from "TC3_0" by adding some behaviors as follows. In TC3_0.1, after the slave is started, it can wait for some time before it starts doing its own work. When the slave finished doing its work, it displays information to the screen of user before being synchronized with the master. In TC3_0.1.1, when the slave finished displaying information to user's screen, it can go back to do its own work. In TC3_0.1.1.1, when the slave finished displaying information to user's screen, it can wait for some time before being synchronized with the master or doing its own work. In TC3_0.1.1.1.1, the slave can do its own work for some time before doing other works.

"TC4_0.1" – "TC4_0.1.1.1.1.1" are modified systems from "TC4_0" by adding some behaviors as follows. In TC4_0.1, when the channel finished providing feedback to the monitor thread, it needs to inform user about the status before being ready to send another message. In TC4_0.1.1, when the channel finished providing feedback to the monitor thread, it can be ready to send another message. In TC4_0.1.1.1, when the channel finished providing feedback to the monitor thread, it can send another message. In TC4_0.1.1.1.1, when the channel finished informing user about the status, it can provide feedback to the monitor thread again. In TC4_0.1.1.1.1.1, when the channel can be ready for sending a new message for some time before it actually sends the next message.

In TC5, although it is the same simple communication channel as TC4, different versions of M_1 are tested as follows. In TC5_0.1, when the channel finished providing feedback to the monitor thread, it needs to inform user about the status before being ready to send another message or providing feedback to the monitor thread again. In TC5_0.2, after sending a message, the channel can wait for some time before providing feedback to the monitor thread. In TC5_0.2.1, when the channel finished providing feedback to the monitor thread, it needs to inform user about the status before being ready to send another message. TC5_0.2.1.1, when the channel finished informing user about the status, it can provide feedback to the monitor thread again. In TC5_0.2.1.1.1, when the channel is ready to send another message, it can provide feedback to the monitor thread.

Finally, similar to TC5, TC6 are tested with different versions of M1 as follows. TC6_0.1 is the same as TC5_0.2.1 in TC5. TC6_0.1.1 is the same as TC5_0.2.1.1 of M_1 in TC5. TC6_0.1.1.1 is the same as TC5_0.2.1.1.1 of M_1 in TC5. In TC6_0.1.1.1.1, the channel provides feedback to the monitor thread for some time before doing other works. In TC6_0.1.1.1.1.1, the channel can be ready for sending a new message for some time before it actually sends the next message.

In Table 6, "EQ", "I", "T", "Time(ms)", and "Mem (B)" are the number of equivalence queries, initial and transition membership queries, time (in milliseconds), and memory (in bytes) required to regenerate an assumption after reusing the first assumption, respectively. $|L(A_{RO})|$ and $|L(A_{RN})|$ are the size of the corresponding languages of the assumptions regenerated by CBAG algorithm and LWAG algorithm, and a minus sign ("-") means that the assumption did not need to be regenerated after M_1 modified because the modified M_1' of M_1 already has $L(M_1') \subseteq L(A_{RN})$. Similar to the cases in the previous experiment, we calculated $|L(A_{RO})|$ and $|L(A_{RN})|$ by counting the number of traces in $|L(A_{RO})|$ and $|L(A_{RN})|$ with the bound shown in the "B" column of Table 5.

From the experimental results shown in Table 6, we can see that when using CBAG algorithm, in 26 out of 28 test cases, we still need to regenerate the assumption after M_1 is modified. In contrast, when using LWAG algorithm, in 26 out of 28 test cases, we do not need to regenerate the assumption after M_1 is modified. This result implies that LWAG algorithm and framework reduce the number of times assumptions must be regenerated after M_1 evolves and shows they can reduce the cost of modular verification of the modified CBS. The memory usage in both cases (using assumptions generated by CBAG algorithm or using assumptions generated by LWAG algorithm) is similar.

6.3. Discussion

Although both LWAG algorithm and the proposed framework are presented for software with two components, the method can be extended to be applied for larger systems (i.e., $M = M_0 \parallel M_1 \parallel ... \parallel M_n$, where $n \geq 2$) as discussed in researches of Chen et al. [13], Hung et al. [32], and Lin et al. [41]. On one hand, the assume-guarantee rule can be applied

recursively as follows: $\{M_0 \parallel A_1 \vDash \pi;\ M_1 \parallel A_2 \vDash A_1;\ ...;\ M_{n-1} \parallel A_n \vDash A_{n-1};\ M_n \vDash A_n\} \Longrightarrow M = M_0 \parallel M_1 \parallel ... \parallel M_n \vDash \pi$. This method of applying LWAG algorithm has a high complexity since many assumptions need to be learned. On the other hand, the given system M can be partitioned into two higher level components $H_0 = M_0 \parallel ... \parallel M_i$ and $H_1 = M_{i+1} \parallel ... \parallel M_n$ with $0 \leq i < n$ to fit the assume-guarantee rule. For a given CBS M, the evolution can occur on some components of M. When applying the proposed framework for M, we divide M into two higher level components H_0 which contains unchanged components and H_1 which contains modified components. The application of the framework for M contains similar steps as described in Section 5 because we only care about the modified component H_1' of H_1.

When carrying out experiments, because of the limitations of test systems and the computing power of the experimental environment, we only performed experiments using some small systems. The results show considerable potential for applying the proposed method to practical systems regarding the following aspects.

- The maximum values of $|X_0|$ and $|X_1|$ are 4. This means that these test cases can represent systems combined from components and properties that have up to $2^4 = 16$ states. As a result, we can verify software systems in which their maximum sizes can be $size\ of\ component\ 0 \times size\ of\ component\ 1 \times (size\ of\ property + 1) = |C_0| \times |C_1| \times |p_{err}| = 2^4 \times 2^4 \times (2^4 + 1) = 4352$ [35], where C_0, C_1, and p are represented by LTS. In addition, we care only about observable actions of the components. This allows us to apply the proposed method to practical systems that have complex internal implementations but have a limited number of observable actions, such as STS [47] and PLC [54]. This experimental results indicate the reliable effectiveness when applying LWAG algorithm and the proposed framework to large scale systems in practice.
- Although only some small test cases were used in the experiment, they clearly show that LWAG algorithm generates weaker assumptions than those generated by CBAG algorithm. These test cases also show that the assumptions generated by LWAG algorithm can reduce the number of times that assumptions must be regenerated to verify of evolving software. Although the time saved each time an assumption does not need to be regenerated is small, our approach can be applied many times during the software life-cycle because change can occur at any time in any phase of the software development process. As a result, the obtained benefit from our approach grows over time. Using this approach, the framework is effective for verifying practical software.
- When testing with a large test case, such as test case 6 ($TC6$), the time needed to regenerate assumptions becomes greater. Consequently, reducing the number of times that an assumption needs to be regenerated would accelerate many software reverification efforts. Once again, this result shows the effectiveness of the proposed framework when applied to evolving software verification.
- The verification complexity depends on both the number of transitions inside M_0 and M_1 and the number of variables in X_0 and X_1 to encode the system. For the four systems TC1, TC2, TC3, and TC4 which have 3 to 4 transitions in both M_0 and M_1, there is not much difference in the verification running time. The reason is that when encoding a given system, each transition will be encoded into one DNF predicate in the transition function while CDNF algorithm complexity depends on the number of DNF and CNF predicates in the Boolean function to be learned. With the same system of the simple communication channel, it is much faster when using two Boolean variables to encode both components M_0 and M_1 (i.e., TC4) than that when using three Boolean variables (i.e., TC5) and when using four Boolean variables (i.e., TC6). The reason is that the complexity of algorithms implemented in the *teacher* depends on the number of Boolean variables in the system under check. This result gives us a suggestion that we should use the minimum number of Boolean variables to encode the system under checking for the best verification speed.
- In the proposed framework shown in Section 5, we assume that M_0 is a type of static framework and that M_1 is a business component subject to change during the software life cycle. Under this hypothesis, we can reuse the weaker assumptions generated by LWAG algorithm in the framework when M_1 is modified. In addition, effectively decomposing a given software application into components is another major problem when working with assume-guarantees specifically and component-based software in general. This problem is outside the scope of this research paper. However, we are aware of the problem and will consider addressing it in future research.

7. Related works

Several existing papers on evolving software verification are relevant to our research [11,13,23,27,31–35,44].

In 2010, Chen et al. proposed a purely implicit solution to the contextual assumption generation problem in assume-guarantee reasoning [13]. However, this paper did not consider the case in which the software component has been modified. Instead, when a component has been modified, the assumption–generation method must be executed again from the beginning to regenerate the assumptions for the entire modified system. In contrast, our paper focuses on the context of component change to improve CBAG algorithm [13]. Our target is to reduce the number of assumption regenerations when the component is modified by generating weaker assumptions that can be reused more often than those generated by CBAG algorithm.

In 2016, He et al. proposed a fine-grained learning technique for regression verification for component-based software [27]. Although He et al.'s technique was an excellent idea and garnered good experimental results, it was different from our paper in the following three aspects.

First, when performing the initial verification for component-based software, the key idea of the fine-grained learning technique was to learn each of the subpredicates of the assumption to be generated in a separated learning instance. The candidate was the combination of all subcandidates and submitted to the *teacher*. Obviously, this approach does not reduce the overall computation cost of the learning progress. Instead, the fine-grained learning technique achieved faster speed due to the simultaneous learning of the subpredicates of the assumption to be generated. It generated the same assumption as the one generated by CBAG algorithm [13]. In contrast, LWAG algorithm is target toward generating a weaker assumption that can be reused more effectively for verification when the software evolves. To generate weaker assumptions, the algorithm needs to process more. Therefore, the complexity of LWAG algorithm is greater than that of CBAG algorithm.

Secondly, when performing regression verification for the modified software, the method proposed by He et al. [27] always needs to generate new assumptions even for small evolutionary changes by regenerating the corresponding subpredicates of the assumption to match the changed subpredicates of the software component. However, our proposed framework does not need to regenerate the assumptions for small changes in the modified component because the assumption generated by our method is weaker than the one generated by Chen's method [13].

Lastly, He et al. assumed that the component models are decomposed into smaller subpredicates [27]; however, that is not an easy task in practice in terms of time or the algorithm complexity required. Moreover, when performing regression verification, the method proposed by He et al. [27] needs to compare each of the subpredicates of the system both before and after change. This is also not easy in practice with regard to time complexity. In contrast, the assumption generation method and the framework for verifying modified software proposed in this paper use the component models directly and effectively.

Groce et al. proposed a method called Adaptive Model Checking (AMC) that used inaccurate and updated models to perform verification as they were refined [23]. Nonetheless, the model used in AMC is the whole system model. Thus, when verifying the modified system, the *state explosion* problem may occur, particularly when checking large-scale systems. Moreover, AMC uses automatons to describe the system under checking. While we share the idea of verification of evolving software with this previous paper, our paper uses an implicit representation of the CBS, focuses only on the modified components and attempts to reuse previous verification results when performing reverification.

Chaki et al. focused on checking component substitutability from the verification viewpoint directly [11]. The paper also proposed an algorithm to verify the evolving system dynamically. We share the motivation of this paper concerning evolving systems. Our proposed framework can be used for all types of change. Moreover, the proposed framework is simpler than the method proposed by Chaki et al. [11]. In addition, we use an implicit representation of the CBS, while Chaki used an automaton representation.

Hung et al. proposed a method to optimally generate the minimized assumption for assume-guarantee reasoning [31–35]. This assumption can be used to recheck a modified system at a much lower cost [31]. However, because the cost to generate minimized assumptions is very high, the method is not practically applicable to large-scale systems [35]. These studies also proposed an efficient framework for reverifying modified CBS and were also based on the idea of reusing the previous verification results to reduce the assumption regeneration cost [32,33]. We share the motivation of these studies to reduce verification cost when rechecking modified systems by reusing the previous verification results. That is, we reuse the previous assumption as the starting point for regenerating the assumption when rechecking a modified system. However, we use weaker assumptions than those generated by CBAG algorithm to reduce the number of times assumptions must be regenerated while Hung used minimized assumptions [32]. In addition, we use implicit system representations while Hung used automata to model them. These differences make our method faster than the one proposed by Hung and also make it applicable to large-scale systems.

In 2014, Menghi proposed an approach to extend classical verification algorithms to consider incomplete and evolving specifications [44]. His paper attempted to ensure that after any change, only the part of the system affected by the changes needed to be rechecked to avoid reverification from scratch. This paper extended various existing modeling formalisms to express incompleteness. We share the idea of reusing the previous verification results when rechecking modified systems to avoid rechecking the entire system from the scratch; however, we focus on using an implicit CBS representation during verification and on reducing the number of times assumptions must be regenerated.

Chaki and Strichman proposed three optimizations to the L^* based automated Assume-Guarantee reasoning algorithm for the compositional verification of concurrent systems [12]. The paper suggested an optimization that uses some information already available to the *teacher* to avoid many unnecessary membership and candidate queries. However, this used a labeled transition system specification and did not consider the software evolution context. We use an implicit software specification, improve the assumption generation method proposed by Chen et al. [13], and apply it in the context of software evolution to provide a greater reduction of the regression verification cost.

8. Conclusion

In this paper, we presented an effective framework for rechecking evolving software using LWAG algorithm with an improved technique for answering membership queries during the assumption learning process. Although LWAG algorithm has a greater time complexity than does CBAG algorithm, it can generate local weakest assumptions to reduce the number of assumption regenerations when rechecking evolving software. An implemented tool and experimental results are also presented that allows comparing both assumption generation algorithms and assumption regeneration processes for evolving

systems. The experimental results show that the improved assumption generation algorithm generates weaker assumptions, at the cost of a longer execution time. However, in the long run, the weak assumptions reduce the cost for reverifying evolving systems. Some discussions concerning the experiments are provided in the paper.

Although the experiments in this study were conducted with only small evolving systems, we plan to apply the algorithm and framework to larger and to practical systems to show their usefulness. With large software in practice, in which the cost of each reverification becomes larger, (as shown in test case $TC4\_v0$ in Table 5), being able to reduce the number of assumption regenerations plays a key role in verifying modified software. In addition, LWAG algorithm generates only the locally weakest assumption among all the possible assumptions generated by the backtracking algorithm, as shown in Lemma 6. For the future work, it is focused on generating the globally weakest assumption. This globally weakest assumption will play an even more important role in reducing the verification cost in an evolving software context. In addition, as discussed in Section 6.3, we are also working on a method that can divide a predefined component-based software into components to effectively apply assume-guarantee verification.

Acknowledgements

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

References

[1] K. Abd Elkader, O. Grumberg, C.S. Păsăreanu, S. Shoham, Automated circular assume-guarantee reasoning with n-way decomposition and alphabet refinement, in: S. Chaudhuri, A. Farzan (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2016, pp. 329–351.
[2] K. Abd Elkader, O. Grumberg, C.S. Păsăreanu, S. Shoham, Automated circular assume-guarantee reasoning, Form. Asp. Comput. 30 (5) (September 2018) 571–595.
[3] R. Alur, L. Fix, T.A. Henzinger, Event-clock automata: a determinizable class of timed automata, Theor. Comput. Sci. 211 (1–2) (January 1999) 253–273.
[4] D. Angluin, Learning regular sets from queries and counterexamples, Inf. Comput. 75 (2) (November 1987) 87–106.
[5] M. Archer, B.D. Vito, C. Muñoz, Developing user strategies in pvs: a tutorial, in: Proceedings of the First International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics (STRATA'03), NASA/CP-2003-212448, 2003.
[6] H. Barringer, D. Giannakopoulou, Proof rules for automated compositional verification through learning, in: Proc. SAVCBS Workshop, 2003, pp. 14–21.
[7] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, Systems and Software Verification: Model-Checking Techniques and Tools, 1st edition, Springer Publishing Company, Incorporated, 2010.
[8] R. Bouchekir, M.C. Boukala, Learning-based symbolic assume-guarantee reasoning for Markov decision process by using interval Markov process, Innov. Syst. Softw. Eng. 14 (3) (September 2018) 229–244.
[9] R. Bouchekir, M.C. Boukala, Toward implicit learning for the compositional verification of Markov decision processes, in: M.F. Atig, S. Bensalem, S. Bliudze, B. Monsuez (Eds.), Verification and Evaluation of Computer and Communication Systems, Springer International Publishing, Cham, 2018, pp. 200–217.
[10] N.H. Bshouty, Exact learning Boolean functions via the monotone theory, Inf. Comput. 123 (1995) 146–153.
[11] S. Chaki, E.M. Clarke, N. Sharygina, N. Sinha, Verification of evolving software via component substitutability analysis, Form. Methods Syst. Des. 32 (3) (June 2008) 235–266.
[12] S. Chaki, O. Strichman, Optimized L*-based assume-guarantee reasoning, Chapter, in: 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. Proceedings, Portugal, March 24 – April 1, 2007, Springer Berlin Heidelberg, Heidelberg, March 2007, pp. 276–291.
[13] Y.-F. Chen, E.M. Clarke, A. Farzan, M.-H. Tsai, Y.-K. Tsay, B.-Y. Wang, Automated assume-guarantee reasoning through implicit learning, in: T. Touili, B. Cook, P. Jackson (Eds.), Computer Aided Verification, in: Lecture Notes in Computer Science, vol. 6174, Springer Berlin Heidelberg, 2010, pp. 511–526.
[14] Y.-F. Chen, A. Farzan, E.M. Clarke, Y.-K. Tsay, B.-Y. Wang, Learning minimal separating dfa's for compositional verification, in: S. Kowalewski, A. Philippou (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 31–45.
[15] E.M. Clarke, E.A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, Workshop, Springer-Verlag, London, UK, UK, 1982, pp. 52–71.
[16] E.M. Clarke, W. Klieber, M. Nováček, P. Zuliani, Model Checking and the State Explosion Problem, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 1–30.
[17] E.M. Clarke, D. Long, K. McMillan, Compositional model checking, in: Proceedings of the Fourth Annual Symposium on Logic in Computer Science, IEEE Press, Piscataway, NJ, USA, 1989, pp. 353–362.
[18] E.M. Clarke Jr., O. Grumberg, D.A. Peled, Model Checking, MIT Press, Cambridge, MA, USA, 1999.
[19] J.M. Cobleigh, D. Giannakopoulou, 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, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 331–346.
[20] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Commun. ACM 18 (8) (August 1975) 453–457.
[21] D.A. Duffy, Principles of Automated Theorem Proving, John Wiley & Sons, Inc., New York, NY, USA, 1991.
[22] A. Farzan, Y.-F. Chen, E.M. Clarke, Y.-K. Tsay, B.-Y. Wang, Extending automated compositional verification to the full class of omega-regular languages, in: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 2–17.
[23] A. Groce, D. Peled, M. Yannakakis, Adaptive model checking, in: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '02, Springer–Verlag, London, UK, 2002, pp. 357–370.
[24] O. Grumberg, D.E. Long, Model checking and modular verification, ACM Trans. Program. Lang. Syst. 16 (3) (May 1994) 843–871.
[25] A. Gupta, K.L. Mcmillan, Z. Fu, Automated assumption generation for compositional verification, Form. Methods Syst. Des. 32 (3) (June 2008) 285–301.
[26] F. He, X. Gao, M. Wang, B.-Y. Wang, L. Zhang, Learning weighted assumptions for compositional verification of Markov decision processes, ACM Trans. Softw. Eng. Methodol. 25 (3) (June 2016) 21.
[27] F. He, S. Mao, B.-Y. Wang, Learning-Based Assume-Guarantee Regression Verification, Springer International Publishing, Cham, 2016, pp. 310–328.
[28] T. Henzinger, Z. Manna, A. Pnueli, Temporal proof methodologies for timed transition-systems, Inf. Comput. 112 (2) (August 1994) 273–337.
[29] T.A. Henzinger, S. Qadeer, S.K. Rajamani, You assume, we guarantee: methodology and case studies, in: A.J. Hu, M.Y. Vardi (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 440–451.

[30] C.A.R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (October 1969) 576–580.
[31] P. Hung, T. Aoki, T. Katayama, A minimized assumption generation method for component-based software verification, Chapter, in: Theoretical Aspects of Computing - ICTAC 2009: 6th International Colloquium. Proceedings, Kuala Lumpur, Malaysia, August 16–20, 2009, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 277–291.
[32] P.N. Hung, T. Aoki, T. Katayama, An effective framework for assume-guarantee verification of evolving component-based software, in: Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops, IWPSE-Evol '09, ACM, New York, NY, USA, 2009, pp. 109–118.
[33] P.N. Hung, T. Katayama, Modular conformance testing and assume-guarantee verification for evolving component-based software, in: Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific, December 2008, pp. 479–486.
[34] P.N. Hung, V.-H. Nguyen, T. Aoki, 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), IEEE RIVF International Conference, February 2012, pp. 1–6.
[35] P.N. Hung, V.H. Nguyen, T. Aoki, T. Katayama, On optimization of minimized assumption generation method for component-based software verification, IEICE Trans. 95-A (9) (2012) 1451–1460.
[36] K. Ji, Y. Liu, S.-W. Lin, J. Sun, J. Dong, T. Nguyen. Cell, A compositional verification framework, in: D. Van Hung, M. Ogawa (Eds.), Automated Technology for Verification and Analysis, in: Lecture Notes in Computer Science, vol. 8172, Springer International Publishing, 2013, pp. 474–477.
[37] D. Kapur, M. Subramaniam, Lemma discovery in automating induction, in: M.A. McRobbie, J.K. Slaney (Eds.), Automated Deduction — Cade-13, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 538–552.
[38] M. Kaufmann, J. Moore, Some key research problems in automated theorem proving for hardware and software verification 98 (01 2004) 181–196.
[39] C.-L. Le, H.-V. Tran, P.N. Hung, On Implementation of the Assumption Generation Method for Component-Based Software Verification, Springer International Publishing, Cham, 2017, pp. 549–558.
[40] S.-W. Lin, É. André, J.S. Dong, J. Sun, Y. Liu, An efficient algorithm for learning event-recording automata, in: T. Bultan, P.-A. Hsiung (Eds.), Automated Technology for Verification and Analysis, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 463–472.
[41] S.-W. Lin, E. André, Y. Liu, J. Sun, J.S. Dong, Learning assumptions for compositional verification of timed systems, IEEE Trans. Softw. Eng. 40 (2) (February 2014) 137–153.
[42] S.-W. Lin, J. Sun, T.K. Nguyen, Y. Liu, J.S. Dong, Interpolation guided compositional verification (t), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), November 2015, pp. 65–74.
[43] J. Magee, J. Kramer, Concurrency: State Models &Amp; Java Programs, John Wiley & Sons, Inc., New York, NY, USA, 1999.
[44] C. Menghi, Verifying incomplete and evolving specifications, in: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, ACM, New York, NY, USA, 2014, pp. 670–673.
[45] W. Nam, P. Madhusudan, R. Alur, Automatic symbolic compositional verification by learning assumptions, Form. Methods Syst. Des. 32 (3) (June 2008) 207–234.
[46] A. Pnueli, In transition from global to modular temporal reasoning about programs, in: K.R. Apt (Ed.), Logics and Models of Concurrent Systems, Springer-Verlag New York, Inc., New York, NY, USA, 1985, pp. 123–144.
[47] P. Poizat, J.-C. Royer, A formal architectural description language based on symbolic transition systems and modal logic, J. Univers. Comput. Sci. 12 (12) (2006) 1741–1782.
[48] J.-P. Queille, J. Sifakis, Specification and verification of concurrent systems in cesar, in: Proceedings of the 5th Colloquium on International Symposium on Programming, Springer-Verlag, London, UK, UK, 1982, pp. 337–351.
[49] R.L. Rivest, 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, ACM, New York, NY, USA, 1989, pp. 411–420.
[50] N. Sinha, E.M. Clarke, Sat-based compositional verification using lazy learning, in: Proceedings of the 19th International Conference on Computer Aided Verification, CAV'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 39–54.
[51] M. Sipser, Introduction to the Theory of Computation, 1st edition, International Thomson Publishing, 1996.
[52] T. Vale, I. Crnkovic, E.S. de Almeida, P.A.d.M. Silveira Neto, Y.a.C. Cavalcanti, S.R.d.L. Meira, Twenty-eight years of component-based software engineering, J. Syst. Softw. 111(C) (January 2016) 128–148.
[53] A. Wijs, T. Neele, Compositional model checking with incremental counter-example construction, in: R. Majumdar, V. Kunčak (Eds.), Computer Aided Verification, Springer International Publishing, Cham, 2017, pp. 570–590.
[54] M. Zhou, H. Wan, R. Wang, X. Song, C. Su, M. Gu, J. Sun, Formal component-based modeling and synthesis for plc systems, Comput. Ind. 64 (8) (October 2013) 1022–1034.