VNU Journal of Science: Comp. Science & Com. Eng., Vol. 32, No. 2 (2016) 31–42

A Framework for Modeling and Modular Verification of Component-Based System Designs[☆]

Chi-Luan Le[1,2,*], Hoang-Viet Tran[1], Pham Ngoc Hung[1]

[1]*Faculty of Information Technology,*
VNU University of Engineering and Technology,
E3 Building, 144 Xuan Thuy Street, Cau Giay, Hanoi, Vietnam
[2]*Faculty of Information Technology,*
University of Transport Technology,
H1 Building, 54 Trieu Khuc Street, Thanh Xuan, Hanoi, Vietnam

Abstract

This paper introduces a framework for modeling and verifying safety properties of component-based systems (CBS) by extracting their models from designs in the form of UML 2.0 sequence diagrams. Given UML 2.0 sequence diagrams of a CBS, the framework extracts regular expressions exactly describing behaviors of the system. From these expressions, the proposed framework then generates accurate operation models represented by labeled transition systems (LTSs). After that, these models are used to modular check whether given designs satisfy required safety properties by using the assume-guarantee reasoning paradigm. This framework is not only useful for modeling and verifying designs at design phase, but also for effectively rechecking the correctness of CBS in the context of software evolution. Implemented tools and experimental results are also presented in order to show the feasibilities and effectiveness of the proposed framework.

Received 01 December 2015, revised 13 January 2016, accepted 14 January 2016

Keywords: Software Modeling, Sequence Diagram Analysis, Assume-Guarantee Verification, Model Checking.

1. Introduction

The specification and verification approaches nowadays play an important role in guaranteeing software quality. The assume-guarantee verification [11] has been considered as a potential method for solving the state space explosion problem when checking of large scale CBSs. It can be applied at both design and implementation phases. However, the current researches in regards to this method often assume that the models of systems under checking are already available. This makes the methods difficult to be applied in practice because generating models for systems is a hard problem. The method presented in [12] had mentioned a way of using the model generated from the design artifacts to check safety properties of the system implementation. However, the paper did not describe in details how to use and what kind of artifacts of design level to use to generate component models that will be used in verification. In [15], the author proposed a way to check the consistency of software designs by a set of consistency rules defined by users. However, the method is not used for verifying system designs against safety properties. In regards to the system verification, the research carried out in [16] also addresses the problem

[☆] This work is dedicated to the 20th Anniversary of the IT Faculty of VNU-UET.
[*] Corresponding author. Email: luanlc@utt.edu.vn

of verifying properties of systems through its given UML 2.0 sequence diagrams. However, that is for each of the separate fragments and properties are written in PPTL. Moreover, the method in [16] has not solved the problem for the whole sequence diagrams when all of the fragments are integrated. Although the mentioned researches have addressed an important part of the verification process, they have not shown a complete method of how to do design verification of CBSs. On the other hand, there are other studies that focus on generating models for CBS. Nevertheless, they have not been integrated with any verification method. The method proposed in [17] is used to generate models from sets of traces by doing experiment on components and bases on the Thompson algorithm [1]. The model generation method in [13] is used to retrieve extended finite state machines from interactive traces. The work presented in [9] generates finite state models from source code of software programs written in Java. While these researches have great contribution in model generation, they have not been integrated the generated models with any verification method. From the above reason, this paper proposes a framework to integrate model generation methods with verification ones in order to be applied in the real software development world. The framework generates regular expressions for the behaviors of CBS from sequence diagrams. It then parses these expressions to create operation models in the form of LTSs that exactly describe the system behaviors. In the end, it applies the assume-guarantee reasoning paradigm to check if the system satisfies a given property. This method of verification prevents us from the state explosion problem. This framework is not only useful in design phase but also in system maintenance when the design is changed. The paper is organized as follows. At first, we present some background definitions which are used in Section 2. An overview of the framework is described in Section 3. Section 4 shows algorithms to generate regular expressions from given sequence diagrams. The mechanism to generate models from the result regular expressions of Section 4 is shown in Section 5. The generated models are then used in automatic verification in Section 6. The implemented tool and experimental results are shown in Section 7. Finally, we conclude the paper in Section 8.

2. Background

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

LTSs. This paper uses *Labeled Transition Systems* (LTSs) to model behaviors of components. Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action unobservable to a component's environment. We use π to denote a special error state. An LTS is defined as follows.

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

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

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

Definition 2. *(Trace). A trace σ of an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is a finite sequence of actions $a_1 a_2 ... a_n$, such that there exists a sequence of states starting at the initial state (i.e., $q_0 q_1 ... q_n$) such that for $1 \leq i \leq n$, $(q_{i-1}, a_i, q_i) \in \delta$, $q_i \in Q$.*

Note 1. *The set of all traces of M is called the language of M, denoted by $L(M)$. Let $\sigma = a_1 a_2 ... a_n$ be a finite trace of an LTS M. We use $[\sigma]$ to denote the LTS $M_\sigma = \langle Q, \alpha M, \delta, q_0 \rangle$ with $Q = \{q_0, q_1, ..., q_n\}$, and $\delta = \{(q_{i-1}, a_i, q_i)\}$, where $1 \leq i \leq n$.*

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

Definition 3. *(Parallel composition operator). The parallel composition between* $M_1 = \langle Q_1, \alpha M_1, \delta_1, q_0^1 \rangle$ *and* $M_2 = \langle Q_2, \alpha M_2, \delta_2, q_0^2 \rangle$, *denoted by* $M_1 || M_2$, *is defined as follows. If* $M_1 = \prod$ *or* $M_2 = \prod$, *then* $M_1 || M_2 = \prod$, *where* $\prod$ *denotes the LTS* $\langle \{\pi\}, Act, \emptyset, \pi \rangle$. *Otherwise,* $M_1 || M_2$ *is an LTS* $M = \langle Q, \alpha M, \delta, q_0 \rangle$ *where* $Q = Q_1 \times Q_2$, $\alpha M = \alpha M_1 \cup \alpha M_2$, $q_0 = (q_0^1, q_0^2)$, *and the transition relation* δ *is given by the following rules:*

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

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

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

Safety LTSs, Safety Property, Satisfiability and Error LTSs.

Definition 4. *(Safety LTS). A safety LTS is a deterministic LTS that contains no* π *states.*

Note 2. *A safety property asserts that nothing bad happens for all time. The safety property p is specified as a safety LTS* $p = \langle Q, \alpha p, \delta, q_0 \rangle$ *whose language* $L(p)$ *defines the set of acceptable behaviors over* αp.

Definition 5. *(Satisfiability). an LTS M satisfies p, denoted by* $M \models p$, *if and only if* $\forall \sigma \in L(M)$: $(\sigma \upharpoonright \alpha p) \in L(p)$, *where* $\sigma \upharpoonright \alpha p$ *denotes the trace obtained by removing from* σ *all occurrences of actions* $a \notin \alpha p$.

Note 3. *When we check whether an LTS M satisfies a required property p, an* error LTS, *denoted by* p_{err}, *is created which traps possible violations with the* π *state.* p_{err} *is defined as follows:*

Definition 6. *(Error LTS). An* error LTS *of a property* $p = \langle Q, \alpha p, \delta, q_0 \rangle$ *is* $p_{err} = \langle Q \cup \{\pi\}, \alpha p, \delta', q_0 \rangle$, *where* $\delta' = \delta \cup \{(q, a, \pi) \mid a \in \alpha p$ *and* $\nexists q' \in Q : (q, a, q') \in \delta\}$.

Remark 1. *The error LTS is complete, meaning each state other than the error state has outgoing transitions for every action in the alphabet. In order to verify a component M satisfying a property p, both M and p are represented by safety LTSs, the parallel compositional system* $M || p_{err}$ *is then computed. If the state* π *is reachable in the compositional system then M violates p. Otherwise, it satisfies p.*

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

Definition 7. *(Assume-guarantee formula/rule). Let M be a component, p be a property, and* $A(p)$ *be an assumption about M's environment. An assume-guarantee formula/rule is a triple* $(\langle A(p) \rangle\ M\ \langle p \rangle)$ *representing the compositional formula* $A(p) || M || p_{err}$, *where M,* $A(p)$, *and* p_{err} *are presented by LTSs.*

Note 4. *We use the formula* $\langle true \rangle\ M\ \langle A \rangle$ *to represent the compositional formula* $M || A_{err}$. *The formula* $\langle A(p) \rangle\ M\ \langle p \rangle$ *is true if whenever M is part of a system satisfying* $A(p)$, *then the system must also guarantee p. In order to check the formula, where both* $A(p)$ *and p are safety LTSs, we compute the compositional formula* $A(p) || M || p_{err}$ *and check if the error state* π *is reachable in the composition. If it is, then the formula is violated, otherwise it is satisfied.*

Definition 8. *(Assumption). Given two models* M_1 *and* M_2, *and a required safety property p,* $A(p)$ *is an assumption if and only if it is strong enough for* M_1 *to satisfy p but weak enough to be discharged by* M_2 *(i.e.,* $\langle A(p) \rangle\ M_1\ \langle p \rangle$ *and* $\langle true \rangle\ M_2\ \langle A(p) \rangle$ *both hold). Equivalently,* $A(p)$ *is an assumption if and only if* $L(A(p) || M_1) \upharpoonright \alpha p \subseteq L(p)$ *and* $L(M_2) \upharpoonright \alpha A(p) \subseteq L(A(p))$.

3. Framework architecture

Figure 1 shows the architecture of the proposed framework. Sequence diagram designs of

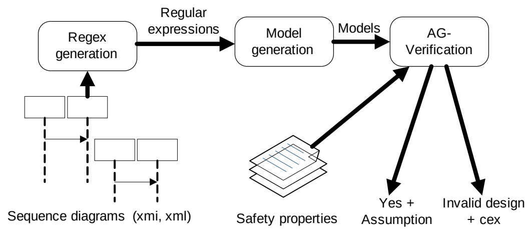

Fig. 1: The proposed framework for verifying designs in the form of sequence diagrams

systems are in the form of an xmi file. They are analyzed to generate corresponding regular expressions. These expressions then are used to generate models. Finally, the framework uses those models and assume-guarantee reasoning paradigm to do modular check to see if given systems satisfy predefined safety properties in the form of LTSs. If designs satisfy properties, the assumption is returned. Otherwise, they violate properties, a counter example is also returned. Details about each of the process are described in Sections 4, 5, and 6.

4. Generating Regular Expression from Sequence Diagrams

In this section, we present algorithms that generate regular expressions of software components' actions from sequence diagrams of design phase. Given a UML 2.0 sequence diagram in the form of xmi file, it is analyzed to get basic fragments such as *opt*, *break*, etc. The corresponding regular expressions of some of them are then generated. These fragments are $opt, break, critical, strict, consider, ignore$. Algorithms for generating regular expressions corresponding to the other fragments of $loop, alt, par/seq$ can be found in [18].

4.1. Analyzing Sequence Diagrams

Given a sequence diagram in the form of xmi file, we use Algorithm 1 to analyze it to have a list of fragments and their relationships.

Algorithm 1 describes the process to analyze the sequence diagram in an xmi file. The result data is an array of $Fragment$ or $Message$ sorted by the time of execution and an array of life line

Algorithm 1: Analyze sequence diagram

```
1  begin
2      create $stack$ with an $Operand$ on top
3      create array $lifelineList$ and array $messageList$
4      forall the element in xmi file do
5          if meet open tag then then
6              switch element do
7                  case $LifeLine$
8                      create new lifeline and add to $lifelineList$; break
9                  case $Fragment$
10                     create new fragment and push to $stack$; break
11                 case $Operand$
12                     create new operand and push to $stack$; break
13                 case $Message$
14                     create new message and add to $messageList$;break
15                 case $EventOccurrence$
16                     create new eventoccurrence and add to the Operand on the top of $stack$; break
17                 case $Constraint$
18                     create new constraint and add to the Fragment on the top of $stack$; break
19             endsw
20         else if meet close tag then
21             if element is Operand then
22                 $op = stack.pop()$
23                 add $op$ to the Fragment on top of $stack$
24             else if element is Fragment then
25                 $fm = stack.pop()$
26                 add $fm$ to the Operand on the top of $stack$
27             end
28         end
29     end
30 end
```

($lifeline$). At first, the algorithm initiates a *stack* that contains an $Operand$ (line 2), this $Operand$ is used to store the array of fragment or message in the data structure. Next, it initiates an array of $LifeLine$ and an array of messages (line 3). When parsing the xmi file, if the algorithm meets an open tag (line 5), it bases on the tag's type to process. If the tag type is $Fragments$ (line 9) or $Operand$ (line 11), add these objects to *stack*. If the tag type is $LifeLine$ (line 7) or $Message$ (line 13), add object to the corresponding array. If the tag is *EventOccurrence* (line 15) or $Constraint$ (line 17), add these objects to the object that is on top of the *stack*. If the algorithm meets a close tag (line 20) that is $Operand$ (line 21) or $Fragment$ (line 24), get these object from the top of *stack* and then add them to the object on the top of *stack*. After reading all of the elements in the xmi file, we have an array of $Fragments$ and events inside operands on the top of *stack*, an array of the $LifeLine$ and an array of messages. The couple of events will be replaced by the corresponding messages.

4.2. Generating Sub-Regular Expressions for opt Fragments

Algorithm 2 describes the regular expression generation process for the *opt* fragment. The *opt* fragment contains only one operand which can be executed or not. Therefore, the regular expression corresponding to the *opt* fragment contains the regular expression of operand concatenate with "|" and λ, where λ is a special character represents the empty regular expression.

Algorithm 2: Generate sub regular expression for *opt* Fragments

```
1 begin
2     create regex is empty
3     regex = regex + operand.getRegex() + | + λ
4     return regex
5 end
```

4.3. Generating Sub-Regular Expressions for break / critical / strict Fragments

Algorithm 3 describes the regular expressions generation process for the *break*, *critical* and *strict* fragments. The *break* fragment is only meaningful when it is embedded in the *loop* fragment. Therefore, the *break*'s regular expression is the concatenation of the operands inside the *break*. The same with the *critical* and *strict* fragments. The fragment *critical* only has meaning when embedded in the *par* fragment. The *strict* fragment describes the sequences of actions. Therefore, the result regular expression includes the concatenation of sub-expressions corresponding to the operands inside the *strict*.

Algorithm 3: Generate sub regular expression for $break/critical/strict$ Fragments

```
1 begin
2     create regex is empty
3     forall the operand in fragment do
4         regex = regex + operand.getRegex()
5     end
6     return regex
7 end
```

4.4. Generating Sub-Regular Expressions for consider Fragments

Algorithm 4 describes the process of generating regular expression for the *consider* fragment. The *consider* fragment contains a list of messages need to be kept. If messages in the *consider* operands are not in this list, they are removed. Line 3 to line 7 is the process of finding and removing messages not in *considerList*. Line 8 to line 10 is the process of creating regular expression after removing unneeded messages. The regular expression of the *consider* fragment consists of the sub-regular expressions corresponding to operands belong to *consider* fragments concatenated to each other.

Algorithm 4: Generate sub regular expression for *consider* Fragments

```
Input : considerList is an array which contains messages that need to be kept
Output: The regular expression corresponding to the consider fragment
1  begin
2      create regex is empty
3      forall the element in consider fragment do
4          if element is message and not in considerList then
5              remove element
6          end
7      end
8      forall the operand in consider fragment do
9          regex = regex + operand.getRegex()
10     end
11     return regex
12 end
```

Algorithm 5: Generate sub regular expression for *ignore* Fragments

```
Input : ignoreList is an array which contains messages that need to be ignored
Output: The regular expression corresponding to the ignore fragment
1  begin
2      create regex is empty
3      forall the element in ignore fragment do
4          if element is message and in ignoreList then
5              remove element
6          end
7      end
8      forall the operand in ignore fragment do
9          regex = regex + operand.getRegex()
10     end
11     return regex
12 end
```

4.5. Generating Sub-Regular Expressions for ignore Fragments

Algorithm 5 describes the process of generating the corresponding regular expression for the *ignore* fragments. The *ignore* fragment contains a list of messages that need to be removed. If messages of operands are included in this list, they need to be removed. The removing process is from line 3 to line 7. Line 8 to line 10 is to generate the corresponding regular expressions of the *ignore* fragments. The resulting regular expression is the concatenation of the sub-regular expressions corresponding to operands.

5. Generating Models from Regular Expressions

From the regular expressions returned by the previous section, we can apply several algorithms to generate the corresponding component models. In our study, we applied three algorithms to generate software models in the form of LTSs from the given regular expressions retrieved from the previous step. These algorithms are: Thompson [1], L^* [4] and CNNFA [5, 7]. Each algorithm has its own advantages and disadvantages. We should consider using which algorithm bases on our specific scenarios.

5.1. Generating Models using Thompson Algorithm

Thompson algorithm is a very simple and easy to understand way to build models of components in the form of NFAs from given regular expressions of observable behaviors. The details of the algorithm can be found in [17, 1]. Given a regular expression R_L, the Thompson algorithm will generate a corresponding $\epsilon - NFA$ as follows:

- If $a \in \Sigma$ is a symbol of the alphabet, then a is an atomic regular expression. The NFA that recognizes the regular language of $\{a\}$ is generated as shown in Figure 2, where i

is the initial state, f is the final state and (i, a, f) is the unique transition of the NFA.

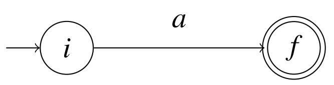

Fig. 2: Generating an NFA that recognizes $\{a\}$.

- Suppose that $N(s)$ and $N(t)$ are non-deterministic finite automata corresponding to the regular expressions s and t respectively, then
  - $(s).(t)$ is a regular expression that represents the language $L(s).L(t)$. The automaton accepting this language is built as shown in Figure 3. The initial state is the initial state of $N(s)$, the final states are the final states of $N(t)$ and the algorithm adds empty transitions from the final states of $N(s)$ to the initial state of $N(t)$.

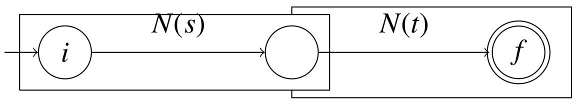

Fig. 3: An NFA recognizes regular expression $(s).(t)$.

  - $(s) + (t)$ is a regular expression that represents the language $L(s) \cup L(t)$. An $\epsilon - NFA$ that corresponds to the regular expression $(s) + (t)$ is built as shown in Figure 4. In this case, the initial state called i and $\epsilon - transitions$ from i to the initial states of $N(s)$ and $N(t)$ are added to the automaton. After that, it adds a final state called f and $\epsilon - transitions$ from the final states of $N(s)$ and $N(t)$ to f. As a result, we have the $\epsilon - NFA$ that is the union of $N(s)$ and $N(t)$.
  - (s^*) is a regular expression that represents the language $L(s^*)$. An $\epsilon - NFA$ that corresponds to the regular expression (s^*) is built as shown in Figure 5. In this case, the initial state is called i. An $\epsilon - transition$ from f to the initial state of i is added to the automaton. As a result, we have the $\epsilon - NFA$ that is the $N(s^*)$.

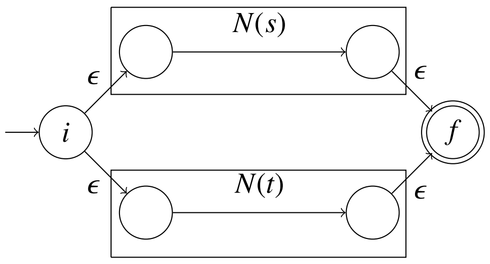

Fig. 4: An NFA recognizes regular expression $(s) + (t)$.

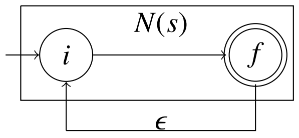

Fig. 5: An NFA recognizes regular expression (s^*).

5.2. Generating Models using L^* Algorithm

The L^* is used to generate the M models that can describe the behaviors of the component C. In order to generate models, the L^* algorithm depends on a Teacher that answers two kinds of question. The first kind is the membership question. With $\sigma \in \Sigma^*$, Teacher answer *true* if $\sigma \in L(C)$ and vice versa. Next, Teacher answers the equivalence query. That is whether the M_i model can describe the whole behavior of the component C or not. If the model can describe the model exactly, M_i becomes the model of C. Otherwise, Teacher provides a counter example cex to L^* to learn again (e.g: $cex \in L(C) \setminus L(M_i)$ or $cex \in L(M_i) \setminus L(C)$) in order to generate new model that can describe the component better.

In order to represent behaviors of models, the L^* algorithm uses the table V, W, T that is defined as follows:

- $V \in \Sigma^*$ is a set of prefixes. Prefixes represent classes or states.
- $W \in \Sigma^*$ is a set of suffixes. Suffixes represent the differences of languages.
- $T : (S \cup S.\Sigma).E \rightarrow \{true, false\}$, where the operator "." means that given two sets

of sequences P and Q, $P.Q = \{pq | p \in P, q \in Q\}$, where pq represents the concatenation of the event sequences p and q. With a string s in Σ^*, $T(s) = true$ means $s \in U$, otherwise $s \notin U$.

Algorithm 6 describes the model generation process using the L^* learning algorithm. The algorithm requires the component (C) and a maximum length of sequence of actions in the component (n). At first, the algorithm initiates the OT with $V = \{\lambda\}$, $W = \{\lambda\}$, $T = T_C$ and $\Sigma = \Sigma_C$ (λ is the empty string) (line 2). Next, the table is updated by using the component C to answer whether a specific action can be performed on the component (line 4). After updating, the algorithm checks whether the table is closed or not. If the table is not closed, va is added to V where $v \in V, a \in \Sigma$ (line 6) and the table is updated again (line 7). After the table updating process, we have a corresponding model candidate that represents the behaviors of the component. The OT table is used by VC algorithm [3] to check whether the corresponding model can represent the behaviors of the given component or not (line 9). If the model can represent the component, that model is returned by the algorithm (line 12). Otherwise, a counter example is provided by VC to the learning process to generate a new better model. The counter example is analyzed to find the smallest suffix that is not in the suffixes set of the OT table (line 14). The found suffix is added to the set of suffixes W. The OT table is then updated and the algorithm L^* generates a new better model (line 4).

Algorithm 6: Generate models using L^* algorithm

```
Input : Component C, maximum length n
1  begin
2      $OT = (V, W, T)$ with $V = \{\lambda\}$,
       $T = T_C, \Sigma = \Sigma_C$
3      while $true$ do
4          Update $OT_i$ by T
5          while $OT_i$ is not closed do
6              add $va$ to $V$ ($v \in V, a \in \Sigma$)
7              update $OT_i$ by T to make it
               closed
8          end
9          $conform = VC(OT_i, C, n)$
10         if $conform = true$ then
11             create LTS $M_i$ from $OT_i$
12             return $M_i$
13         else
14             $v'$ = minimum suffix(conform)
               that is not in $W$
15             Add $v'$ to $M_i$ of $OT_i$
16         end
17     end
18 end
```

5.3. Generating Models using CNNFA algorithm

The key idea when using the CNNFA algorithm to generate models corresponding to regular expressions is that it uses an algorithm to parse the given regular expression into basic and non-basic blocks. A basic block is a valid sub-regular expression that contains at least one symbol in the alphabet. Non-basic blocks are parts of the regular expression separated by basic blocks. While doing that, it constructs the CNNFA representations for basic blocks and perform reduction steps (from line 4 to line 20). When the algorithm halts, if there is only one CNNFA representation, we can build the corresponding models for the given regular expression. Otherwise, the given regular expression is not valid. The algorithm uses a stack (line 1) of elements, each of them is either a symbol from R, or a record N_p that stores a CNNFA representation of the corresponding sub-regular expression. Detailed information about the models generation process using CNNFA algorithm can be found in [19]. The parsing algorithm is shown in algorithm 7.

5.4. Discussion

From the details of the above algorithms when generating models, we can see that the generated models are not optimal. We need to perform additional tasks to optimize the generated models. These tasks are converting

Algorithm 7: Generate models using CNNFA algorithm

```
 1: Initialize the stack to empty.
 2: for each input symbol $c$ in a left-to-right scan through $R$ do
 3:   Push $c$ onto the stack.
 4:   repeat
 5:     if topmost elements of the stack = $\lambda$ then
 6:       Replace by CNNFA representation of $\lambda$.
 7:     else if topmost elements of the stack = $a$, an alphabet symbol then
 8:       Replace by CNNFA representation of $a$.
 9:     else if topmost elements of the stack = $N_J|N_K$ then
10:       Replace by CNNFA representation of $N_{J|K}$.
11:     else if topmost elements of the stack = $N_J N_K$ then
12:       Replace by CNNFA representation of $N_{JK}$.
13:     else if topmost elements of the stack = $N_J^*$ then
14:       Replace by CNNFA representation of $N_{J^*}$.
15:     else if topmost elements of the stack = $(N_J)$ then
16:       Replace by $N_J$.
17:     else
18:       break;
19:     end if
20:   until the above steps can no longer be applied
21: end for
```

models from NFAs to DFAs, then minimizing the returned DFAs to have the optimal models using Hopcroft algorithm [2]. You can also notice that the result models are not LTSs while the required inputs of the assumption generation process are LTSs. We notice that if the state of the component is accepting state every time an action is performed, then all states of the generated models are accepting states. Therefore, those models are LTSs. We can use them in the assumption generation process. Another important point here is that in [17], the generation process is limited by a *MaxLength* represent for the longest testable trace against the component under checking. Generally, using Thompson algorithm [1] to parse regular expressions to generate the corresponding models is not limited by any *MaxLength*. Therefore, in Table 3, we don't have any *MaxLength* information for the model generation method using Thompson algorithm.

6. Assume-Guarantee Verification of Component-Based Software

Let $M_1, M_2, ..., M_n$ be models of the system under checking. These models are generated from Section 5. We need to verify whether the system satisfy a predefined safety property p or not. In this paper, we use assume-guarantee reasoning approach proposed in [11, 14] to do this (e.g., to check the formula $M \models p$, where $M = M_1||M_2||...||M_n$).

For this purpose, the models are divided into two classes (e.g., fixed and extensional components). Let $M_1, M_2, ..., M_i$ be fixed components and $M_{i+1}, ..., M_n (0 < i < n)$ be extensional components, $M_f = M_1||M_2||...||M_i$ and $M_e = M_{i+1}||...||M_n$ are compositional models of the fixed and extensional components, respectively. These compositional models and the property p are inputs of the assume-guarantee verification method in order to check the system.

The goal of the assume-guarantee verification method is to verify whether the system satisfies the property p *without composing* M_f with M_e. For this purpose, an assumption $A(p)$ is generated by applying the L* learning algorithm [4, 6] such that $A(p)$ is strong enough for M_f to satisfy p but weak enough to be discharged by M_e (i.e., $\langle A(p)\rangle\ M_f\ \langle p\rangle$ and $\langle \text{true}\rangle\ M_e\ \langle A(p)\rangle$ both hold, called assume-guarantee rules) [11, 14]. From these assume-guarantee rules, this system satisfies p without verifying on the whole system. In order to obtain such appropriate assumptions, this method applies the assume-guarantee rules in an iterative process presented in Figure 6. At each iteration i, a candidate assumption A_i is produced based on some knowledge about the system under checking and the results of the previous iterations. The following two steps of the assume-guarantee rules are then applied. Step 1 checks whether M_f satisfies p in an

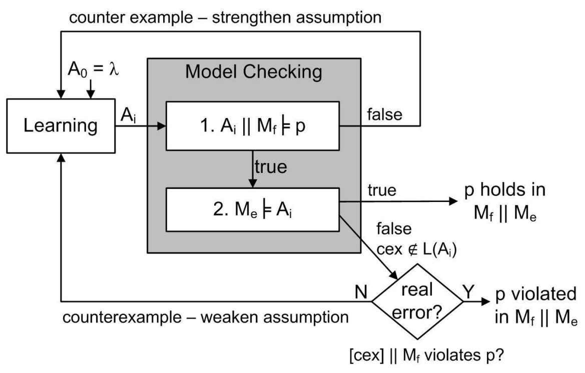

Fig. 6: Framework for the L*-based assumption generation.

environment that guarantees A_i by computing the formula $\langle A_i\rangle\ M_f\ \langle p\rangle$. If the result is $false$, it means that this candidate assumption is *too weak* for M_f to satisfy p. The candidate assumption A_i therefore must be strengthened with the help of the produced counterexample *cex*. Otherwise, the result is *true*. In this case, A_i is strong enough for the property to be satisfied. Then the step 2 is applied for checking whether the component M_e satisfies A_i by computing the formula $\langle true\rangle$ M_e $\langle A_i\rangle$. If this step returns *true*, the property p holds in the compositional system $M_f||M_e$ and the algorithm terminates. Otherwise, this step returns $false$. In this case, a further analysis is required to identify whether p is indeed violated in the system $M_f||M_e$ or the candidate A_i is too strong to be satisfied by M_e. Such analysis is based on the produced counterexample *cex*. For the purpose, the L* algorithm must check whether the counterexample *cex* belongs to the unknown language $U = L(A_W)$, where A_W is the weakest assumption which restricts the environment of M_f no more and no less than necessary for p to be satisfied [10]. If it does not, the property p does not hold in the system $M_f||M_e$. Otherwise, A_i is too strong for M_e to satisfy. The consequence of this is the candidate assumption A_i must be weakened (i.e., behaviors must be added with the help of *cex*) in the next iteration $i + 1$. A new candidate assumption may of course be too weak, and therefore the entire process must be repeated.

7. Experimental Results

In order to show the correctness and feasibility of the proposed framework, we implemented tools to support it. We have tested the method for several systems [8] that contain typical fragments in sequence diagrams until generating the corresponding assumptions. The regular expression generation time is shown in Table 1.

Table 1: Regular expression generation time

No.	System	Time (ms)
1	Mod_channel M1	2.0
2	Mod_channel M2	2.0
3	Mod1 M1	4.0
4	Mod1 M2	40.0
5	Mod2 M1	5.0
6	Mod2 M2	4.0
7	Read_Write M1	2.0
8	Read_Write M2	2.0
9	Simple_channel M1	1.0
10	Simple_channel M2	2.0
11	Two_channel M1	1.0
12	Two_channel M2	2.0
13	GasOverControler	9.0

We then test the model generation process by using the three algorithms of L^*, Thompson, CNNFA. The generation time is presented in the table 2. The size of generated models is shown in the column $|M|$. The columns $|\delta|$ shows the number of transitions in generated models. The generated time (in milliseconds) is shown in the column $Time(ms)$. The *maxlength* in case of generating models using L^* methods is shown in the column *MLen*. “Out” in the columns $Time$ means “Out of memory”, this is the case we could not generate the model using the corresponding algorithm.

From Table 2, we have the following observations:

- Using these testing systems, generating models using Thompson algorithm is faster than the other two methods using L^* and CNNFA algorithm.

Table 2: Model generation time

No.	Test data	L^*				Thompson			CNNFA		
		$\|M\|$	$\|\delta\|$	$MLen$	$Time$	$\|M\|$	$\|\delta\|$	$Time$	$\|M\|$	$\|\delta\|$	$Time$
1	Mod_channel M1	4	3	3	01.44	3	3	00.17	3	3	00.39
2	Mod_channel M2	5	5	4	20.29	3	4	00.26	3	4	11.68
3	Mod1 M1	6	6	3	16.09	5	6	00.75	5	6	26.61
4	Mod1 M2	7	8	4	53.00	5	7	00.83	5	7	233.51
5	Mod2 M1	6	6	3	15.81	5	6	00.75	5	6	76.79
6	Mod2 M2	7	9	4	48.41	5	8	01.02	5	8	421.91
7	Read_Write M1	4	3	3	11.63	3	3	00.24	3	3	00.59
8	Read_Write M2	4	3	3	14.55	3	3	00.20	3	3	00.74
9	Simple_channel M1	4	3	3	11.06	3	3	00.22	3	3	01.73
10	Simple_channel M2	4	3	3	15.05	3	3	00.20	3	3	00.74
11	Two_channel M1	6	6	3	16.06	5	6	00.80	5	6	25.59
12	Two_channel M2	6	6	3	18.55	5	6	00.76	5	6	27.66
13	GasOverControler	-	-	9	Out	6	10	66.28	7	14	4,668.43

Table 3: Assumption Generation Result

No.	System	Verification result	Time (ms)
1	Read_Write	acquireRead.acquireWrite	05.50
2	Mod1	OK	13.58
3	Mod2	OK	05.68
4	Mod_channel	in.send.send.in	00.54
5	Simple_channel	OK	09.21
6	Two_channel	OK	02.26
7	GasOverControler	OK	13.11

- With the big system (GasOverControler), using L^* algorithm cannot generate the models of the system due to out of memory.
- Using the L^* algorithm to generate the model of system is limited by the *maxlength* of the traces recognized by the models.

The time of the assumption generation process is shown in Table 3.

8. Conclusion

We have presented a framework for automated design verification for component-based software. The method generates regular expressions from one of the outputs of the design phase (sequence diagrams). Models corresponding to these regular expressions are then generated. These models are used to verify whether the design satisfies the predefined property or not. The whole process can be re-executed when the design is changed. Experimental result shows that this method is feasible with the time of the verification process.

Although the proposed framework can help us to automatically verify system designs in the form of sequence diagrams, it still contains several issues. The first issue is that it is still slow when testing with large systems. The second is that the models generated and used during verification is in the form of LTSs. This is only one kind of model specification. Currently, the framework is not for other kinds. Last but not least, the framework is only applied for safety properties. What about liveness and fairness ones. Besides, the framework can be extended to generate test paths, test cases and help testing automatically. It can be very helpful for such organizations that not have much testing resources.

We are finding a way to apply the method to some practical and larger systems to prove its effectiveness. We are also extending the method using other kinds of output of design phase (e.g., class diagrams, state-chart diagrams, etc.) so that the given system can be verified in all aspects of design automatically.

Acknowledgments

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

References

[1] K. Thompson, "Regular expression search algorithm", Communications of the ACM 11:6 (1968) 419-422.

[2] J. E. Hopcroft, "An nlogn algorithm for minimizing states in a finite automaton", Tech. Report, Stanford University, Stanford, CA, USA, 1971.

[3] T. S. Chow, "Testing software design modeled by finite-state machines", IEEE Transactions on Soft. Engineering, vol. 4(3), pp. 178–187, 1978.

[4] D. Angluin, "Learning regular sets from queries and counterexamples", Information and Computation, vol. 75, no. 2, pp. 87-106, Nov. 1987.

[5] C. Chang, "From regular expressions to DFA's using compressed NFA's", Ph.D. Thesis, New York University, New York, 1992.

[6] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences", Information and Computation, vol. 103, no. 2, pp. 299-347, April 1993.

[7] C. Chang, R. Paige, "From regular expressions to DFA's using compressed NFA's", Theoretical Computer Science 178, pp. 1-36, 1997.

[8] J. Magee and J. Kramer, Concurrency: "State Models & Java Programs", John Wiley & Sons, 1999. The MIT Press, 1999.

[9] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby and Hongjun Zheng, "Bandera: extracting finite-state models from Java source code", Proceedings of the 2000 International Conference on Software Engineering, pp. 439-448d, 2000.

[10] D. Giannakopoulou, C. Pasareanu, and H. Barringer, "Assumption generation for software component verification", Proc. 17th IEEE Int. Conf. on Automated Softw. Eng., pp. 3–12, Edinburgh, UK, Sept. 2002.

[11] J. M. Cobleigh, D. Giannakopoulou, C. S. Pasareanu, "Learning Assumptions for Compositional Verification", Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 331–346, 2003.

[12] Giannakopoulou, Dimitra and Pasareanu, Corina S. and Cobleigh, Jamieson M., "Assume-Guarantee Verification of Source Code with Design-Level Assumptions", IEEE Computer Society, Proceedings of the 26th International Conference on Software Engineering, pp. 211-220, 2004.

[13] D. Lorenzoli, L. Mariani and M. Pezzè, "Automatic generation of software behavioral models", ACM, Proceedings of the 30th international conference on Soft. engineering, pp. 501-510, 2008.

[14] P. N. Hung, T. Aoki and T. Katayama, "Modular Conformance Testing and Assume-Guarantee Verification for Evolving Component-Based Software. IEICE Trans. on Fundamentals", Special Issue on Theory of Concurrent Systems and Its Applications, Vol. E92-A, No.11, pp. 2772-2780, 2009.

[15] Egyed, A., "Automatically Detecting and Tracking Inconsistencies in Software Design Models", Software Engineering, IEEE Transactions on , vol.37, no.2, pp.188-204, March-April 2011.

[16] Zhang Chen, Duan Zhenhua, "Specification and Verification of UML2.0 Sequence Diagrams using Event Deterministic Finite Automata", pp.41-46, IEEE Computer SocietyWashington, DC, USA 2011.

[17] L. B. Cuong and P. N. Hung, "A Method for Generating Models of Black-box Components", 4th International Conference on Knowledge and Systems Engineering, IEEE Computer Society Press, pp. 177-222, 2012.

[18] H.M. Duong, L.K. Trinh and P. N. Hung, "An Assume-Guarantee Model Checker for Component-Based Systems", The 10th IEEE-RIVF International Conference on Computing and Communication Technologies, pp. 22–26, IEEE Computer Society Press, 2013.

[19] Tran, Hoang-Viet and Le, Chi-Luan and Nguyen, Quang-Trung and Ngoc Hung, Pham, "An Efficient Method for Automated Generating Models of Component-Based Software", Knowledge and Systems Engineering, Springer International Publishing, volumn 326, pp. 499-511, 2015.