Vol.2009-SE-165 No.7
2009/7/3

Regular Paper

Towards Integrating Adaptation and Model Checking for Software Components

Hsin-Hung Lin[†1] and Takuya Katayama[†1]

This paper introduces a framework which converts adaptation for software components into a model checking problem so that one can design/generate adaptor and also perform verification with specifications. In this framework, software components are modeled by interface Büchi automata to capture the input/output protocol and continuous running behavior, and an adaptor is modeled by pushdown automaton to capture the infinite capacity of storing received messages. The adaptation with model checking includes two stages: mismtach detection and adaptor generation. On the first stage, system behavior of a composable components is computed by synchronous composition. LTL model checking is applied with given behavior mismatch property to detect whether mismtaches exist. On the second stage, a special adaptor called "coordinator" is build for the system to guide the adaptor generation with counterexample. Also, a simple example is used for demonstration.

1. Introduction

Component-Based Software Engineering(CBSE) is a specific field of software engineering which deals with software composed of interactive and heterogeneous processes, called software components, or components in short. Software components are usually considered as software entities such as subsystems, objects, or web services. CBSE aims to put software components work together, especially when some components are evolved or reused in new design. For the issue of reuse, coordination and adaptation are two typical approach of CBSE. Coordination model, such as CORBA and J2EE,, specifies protocols of components and/or control strategies for communications. Adaption, which is relative new to coordination, generates an adaptor, a midiate component, to direct interactions of components. Compare to coordination, adaptation is simpler and does no change to components. This is very helpful when having some components reused or provided by third party vendor such that they can not be modified in order to satisfy a new design.

Besides reuse, another important issue of CBSE is verfication. Since software protocols are usually complicated, plus large scale software systems are demanded, one always wants to know if a design of a component-based software will work together as specified. Among techniques of verification of software systems, software model checking is relatively popular. Given a abstract model of a component-based software, software model checking can automatically verify specifications represented in temporal logic. It is also very convenient that many tools support automation of software model checking such as NuSMV and SPIN.

It is very interesting and useful if adaptation and verification of a component-based software can be done in one approach. This research aims to propose a framework for component adaptation with model checking. The framework includes following parts:

(1) Formalization of software components and adaptor: behavior of software components and adaptor are represented by interface Büchi automata and pushdown automaton respectively. Synchronous composition is applied to build the behavior of system without or with an adaptor.

(2) Defining component mismatch with LTL model checking: Detection of component mismtaches over given properties can be defined as a model LTL checking problem on LTL formulas representing these properties.

(3) Adaptor generation with model checking: a special adaptor called coordinator is used to generate adaptor from counterexample.

Fig 1 shows an overview of the approach.

The structure of the rest of this paper is as follows: section 2 gives the definitions of behavior interfaces for modeling components, then define the mismatch detection problem as a LTL model checking problem; section 3 gives definition of behavior model of adaptor and shows how this framework generates adaptor with the guidance of coordinator; section 4 gives a simple example for detailed demonstration of how the framework works; the remaining sections discuss about relative work and conclusions.

†1 School of Information Science, Japan Advanced Institute of Science and Technology

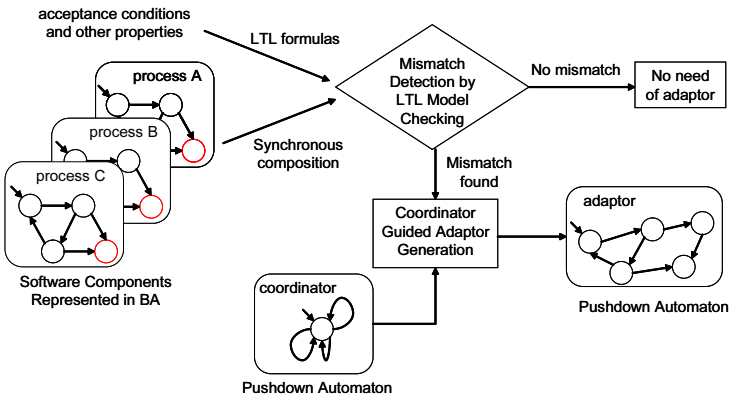

Fig. 1 Overview of approach

2. Behavior Interfaces and Mismatch Detection

This section shows how mismtach detection of component-based software system is treated as a LTL model checking problem. The first thing is to define the model of behavior of software components, then system behavior can be computed by synchronous composition. We then give the LTL definitions for behavior mismatch which is the basic and minimum condition has to be satisfied.

2.1 Behavior Interfaces

The behavior interfaces is essential to adaptation for software components. Researches about adaptation usually use LTS for modeling because of the simplicity and full support of theories and tools. The demerit of using LTS is that messages has to be carefully treated so that input and output can be distinguished clearly, which leads to more complexity in formalization. In this paper we try to capture the characteristic of input and output behavior more explicitly and introduces Interface Büchi automata as the model of software components in this framework. The decription in definition and compatibility constraint are inspired by the work of the interface automata[4] and extend the idea for multiple components more than only two. Idea of Büchi automata is also used here for capture non-stoping characteristics of software components.

Definition 1 (Interface Büchi automata) An interface Büchi automaton is defined as a 7-tuple:

$$P = (Q, q^0, A^I, A^O, A^H, \Delta, F)$$

where

- Q: finite set of states.
- $q^0 \in Q$: initial state.
- A^I: finite set of input alphabets.
- A^O: finite set of output alphabets.
- A^H: finite set of internal alphabets.
- $\Delta \subseteq Q \times A \times Q$: set of transition relations, where $A = A^I \cup A^O \cup A^H$
- $F \subseteq Q_i$: finite set of final states.
- Acceptance condition of interface Büchi automata is same as usual Büchi automata

In this framework, a component-based software given a set of components has to satisfy the compatibility condition. Compatibility means that the given set of software components are composable, which also says that these componsnts form a a closed system. In this system, no messages are being sent to or received from outside the system.

Definition 2 (Compatibility) Given a set of components represented in interface Büchi automata

$$P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, F_i)$$

where $i \in [1..n]$, the components are composable if

- $A_i^I \cap A_i^O = \emptyset$
- $A_i^I \cap A_j^I = \emptyset,\ i \neq j$
- $A_i^O \cap A_i^O = \emptyset,\ i \neq j$
- $\bigcup_i A_i^I = \bigcup_i A_i^O$
- $\bigcup_i A_i^H \cap \bigcup_i A_i^I = \emptyset, \quad \bigcup_i A_i^H \cap \bigcup_i A_i^O = \emptyset$

Given a set of software components that are composable, we can now compute the overall behavior by synchronous composition. The result of composition is a generalized Büchi automaton with only internal alphabets which are union of all input/output and internal alphabets of components.

Definition 3 (Synchronous composition) Synchronous composition of a set of composable components represented in interface Büchi automata $P_i =$

$$
\begin{aligned}
\Delta = \{ & \\
& \{((q_1, \ldots, q_i, \ldots, q_j, \ldots, q_n), a, (q_1, \ldots, q'_i, \ldots, q'_j, \ldots, q_n)) \mid \\
& (q_1, \ldots, q_i, \ldots, q_j, \ldots, q_n),\ (q_1, \ldots, q'_i, \ldots, q'_j, \ldots, q_n) \in Q \wedge \\
& (q_i, a, q'_i) \in \Delta_i \wedge (q_j, a, q'_j) \in \Delta_j \wedge \\
& (a \in A_i^I \wedge a \in A_j^O) \vee (a \in A_i^O \wedge a \in A_j^I)\} \\
& \cup \\
& \{((q_1, \ldots, q_i, \ldots, q_n), a, (q_1, \ldots, q'_i, \ldots, q_n)) \mid \\
& (q_1, \ldots, q_i, \ldots, q_n),\ (q_1, \ldots, q'_i, \ldots, q_n) \in Q \wedge \\
& (q_i, a, q'_i) \in \Delta_i \wedge a \in A_i^H\} \\
\}
\end{aligned}
$$

Fig. 2 Def. of transitions of synchronous composition of interface Büchi automata

$(Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, F_i)$, where $i \in [1, n]$, is a generalized Büchi automaton

$$S = (Q, q^0, A, \Delta, G)$$

where

- $Q = Q_1 \times \ldots \times Q_i \times \ldots \times Q_n$: finite set of states.
- $q_0 = (q_1^0,\ \ldots, q_i^0,\ \ldots, q_n^0)$: initial state.
- $A = \bigcup_i A_i$: finite set of alphabets.
- $\Delta \subseteq Q \times A \times Q$: set of transition relations.
  The transition relations are defined in fig 2.
- $G = \{F_1 \times Q_2 \times \ldots \times Q_n,\ \ Q_1 \times F_2 \times \ldots \times Q_n,\ \ \ldots,\ \ Q_1 \times \ldots \times Q_{n-1} \times F_n\}$: sets of final states.

Generalized Büchi automata are only different from normal Büchi automata in acceptance condition. Finite states of a generalized Büchi automaton is defined as a set of sets computed from each final states of a component times states of other components. Thus an accepted trace has to visit each set of final states infinitely often.

Definition 4 (Acceptance Condition of GBA) For a generalized Büchi automaton $B = (Q, q^0, A, \Delta, G)$, and a trace $\sigma = s_0 s_1 s_2 \ldots$, where $s_0 = q^0$ and $\forall i \geq 0.\ \exists a_i \in A.\ (s_i, a_i, s_{i+1})$, The acceptance condition of trace σ in B is:

$$\forall F_i \in G.\ \forall K \geq 0.\ j \geq K,\ s_j \in F_j$$

Since the acceptance condition for generalized Büchi automata is a little complicate, one always wants to use equivalent Büchi automata for further use. Generally, a degeneralization process for GBA can build equivalent Büchi automata. The idea is to generate as many copies as the number of components and remark the final states as corresponding set of final states, then redirect transitions from i-th copy to corresponding target state in next copy. So we can have a equivalent Büchi automaton that accepts same language as original generalized one.

Definition 5 (Degeneralization of generalized Büchi automata) For a generalized Büchi automaton $B = (Q, q^0, A, \Delta, G)$, where $F = \{F_1, F_2, \ldots, F_n\}$. There is an equivalent Büchi automaton $B' = (Q', q'^0, A, \Delta', F)$, where

- $Q' = Q \times \{1, 2,\ \ldots, n\}$
- $q'^0 = (q^0, 1)$
- $F = F_1 \times \{1\}$
- Δ' is defined as follows:
  $((q_1, i), a, (q_2, j)) \in \Delta'$ iff
  $(q_1, a, q_2) \in \Delta$ and
  $j = i$ if $q_1 \notin F_i$, $\quad j = (i\ mod\ k) + 1$ if $q_1 \in F_i$

By now, we have defined the behavior interface of software components and their synchronous composition. After the process of degeneralization, we have the behavior of the system as a Büchi automaton.

2.2 Mismatch Detection by Model Checking

Given a component-based software system composed of a set of components, we can now define the model checking problem of the system. Before proceeding to mismatch detection, we first review the LTL model checkign problem: Given model M and an infinite execution trace $\sigma = s_0 s_1 \ldots$ of M, with a set of atomic propositions AP and a labeling function $L : S \to 2^{AP}$, model checking a LTL formulas composed of atomic propositions in AP has the following semantics:

- $\sigma \models p$ iff $p \in L(s_0)$
- $\sigma \models \neg\varphi$ iff $\neg(\sigma \models \varphi)$
- $\sigma \models \varphi_1 \wedge \varphi_2$ iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$
- $\sigma \models \varphi_1 \vee \varphi_2$ iff $\sigma \models \varphi_1$ or $\sigma \models \varphi_2$
- $\sigma \models \Box\varphi$ iff $\forall i \geq 0, \sigma^i \models \varphi$
- $\sigma \models \Diamond\varphi$ iff $\exists i \geq 0, \sigma^i \models \varphi$
- $\sigma \models \varphi_1 U \varphi_2$ iff $\exists i \geq 0, \sigma^i \models \varphi_2$ and $\forall 0 \leq j < i, \sigma^j \models \varphi_1$
- $\sigma \models X\varphi$ iff $\sigma^1 \models \varphi$

Note that φ, φ_1, φ_2 are arbitary LTL formulas and σ^i means an execution trace starting from i-th state of σ. Operator $\Box$ means "globally" and $\Diamond$ means "eventally" and X means "next". Based on above semantics, LTL model checking the model M can be defined as follows:

$$M \models \varphi \;\; iff \;\; foralltraces \sigma in M, \sigma \models \varphi$$

LTL formulas have a convenient property that one can build a Büchi automaton which accepts traces of it, thus there is a automata-theoretic approach for LTL model checking including following steps[1)]:

(1) Build the Büchi automaton $B_{\neg\varphi}$ for $\neg\varphi$.
(2) Cmpute product of M and $B_{\neg\varphi}$. The resule accepts $\Sigma_M \cap \Sigma_{\neg\varphi}$.
(3) Check if the product accepts any sequence which is a counter example.

In the case of software components in this framework, the LTL model checking problem is quite simple that the model M is the degeneralized synchronous composition of components, a Büchi automaton. The next problem becomes what properties to check to detect mismatches. In this framework, mismatches depends on specification, but one least property is to guarantee the continuous execution of the system. Thus we define the property of basic mismatch "Behavior Mismatch" as follows:

Definition 6 (Behavior mismatch) Given the degeneralized synchronous composition of a set of composable software components as a Büchi automaton, Property of behavior mismatch is to define an atomic proposition p_{bmis} and a labeling function

$$L(s) : \{p_{bmis} | s \in F\}$$

and write in the followubg form of LTL formula:

$$\Box\Diamond p_{bmis}$$

Then we can check any property with this basic property for a given set of software components. A returned counterexample indicates that mismatches exist, and we can proceed to adaptor generation.

3. Adaptor Generation

3.1 Adaptor

When mismatches are detected, an adaptor is needed. An adaptor is a mediate process that coordinates communication of components. Basic idea is that components are communicating through adaptor so messages sent by a component is first received by adaptor and stored, then being sent by adaptor to target component. In this framework we consider the following characteristics of an adaptor:

(1) An adaptor only receives messages from other components and only sends messages that are received. This requires that an adaptor generates no message.
(2) Any message received by an adaptor is expected to be sent immediately or later. This requires that an adaptor has to store received message for later delivery.

Based on above consideration, a simple storage is a stack and we then choose pushdown automaton as the model of behavior of an adaptor. Further more, because of no message generation, an adaptor only has two types of transitions: push and pop, which perform receive then store and send out respectively.

Definition 7 (Adaptor) Given a set of composable interface Büchi automata $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, F_i)$, $i \in [1, n]$. An adaptor is a pushdown automaton

$$D = (Q_D, q_D^0, A_D, \Gamma, z, T)$$

where

- Q_D: finite set of states.
- q_D^0: initial state.
- $A_D = \bigcup_i A_i^I = \bigcup_i A_i^O$: finite set of alphabets.
- $\Gamma = A_D \cup \{z\} \cup \{\epsilon\}$: finite set of stack symbols.
- z: stack start symbol representing bottom of stack.
- $T \subseteq (Q_D \times A_D \times \Gamma) \times (Q_D \times \Gamma^*)$: set of transition relations.
  T is defined as follows:
  $< p, a, \gamma > \hookrightarrow < p', a\gamma >$: push
  $< p, a, \gamma > \hookrightarrow < p', \epsilon >$: pop
  where $a \in A_D$, $\gamma \in \Gamma$

Note that stack symbols are same as alphabets of an adaptor except special symbols. The transition rules only show how the head of stack is replaced but not regarding the whole content of stack. The system behavior can be similarly obtained by synchronous composition of software components with an adaptor.

$$
\begin{aligned}
T' = \{ & \\
& \{((q_1,\ldots,q_i,\ldots,q_n,q_D),a,\gamma) \hookrightarrow ((q_1,\ldots,q'_i,\ldots,q_n,q'_D),a\gamma) \mid \\
& (q_1,\ldots,q_i,\ldots,q_n,q_D),\ (q_1,\ldots,q'_i,\ldots,q_n,q'_D) \in Q \wedge \\
& (q_i,a,q'_i) \in \Delta_i \wedge a \in A_i^O \wedge (q_D,a,\gamma) \hookrightarrow (q'_D,a\gamma) \in T\} \\
& \cup \\
& \{((q_1,\ldots,q_i,\ldots,q_n,q_D),a,\gamma) \hookrightarrow ((q_1,\ldots,q'_i,\ldots,q_n,q'_D),\epsilon) \mid \\
& (q_1,\ldots,q_i,\ldots,q_n,q_D),\ (q_1,\ldots,q'_i,\ldots,q_n,q'_D) \in Q \wedge \\
& (q_i,a,q'_i) \in \Delta_i \wedge a \in A_i^I \wedge (q_D,a,\gamma) \hookrightarrow (q'_D,\epsilon) \in T \wedge a = \gamma\} \\
& \cup \\
& \{((q_1,\ldots,q_i,\ldots,q_n,q_D),a,(q_1,\ldots,q'_i,\ldots,q_n,q_D)) \mid \\
& (q_1,\ldots,q_i,\ldots,q_n,q_D),\ (q_1,\ldots,q'_i,\ldots,q_n,q_D) \in Q \wedge \\
& (q_i,a,q'_i) \in \Delta_i \wedge a \in A_i^H\} \\
\}
\end{aligned}
$$

Fig. 3 Def. of transitions of synchronous composition of interface Büchi automata with adaptor

The degeneralization is also similar to previous definition.

Definition 8 (Synchronous composition with adaptor) Given a set of composable components represented in interface Büchi automata $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, F_i)$, $i \in [1,n]$, and an adaptor $D = (Q_D, q_D^0, A_D, \Gamma, z, T)$. The synchronous composition is a generalized pushdown Büchi automaton

$$S = (Q, q^0, A, \Gamma, z, T', G)$$

where

- $Q = Q_1 \times \ldots \times Q_i \times \ldots \times Q_n \times Q_D$: finite set of states.
- $q_0 = (q_1^0, q_2^0, \ \ldots, q_n^0, q_D^0)$: initial state.
- $A = \bigcup_i A_i^I = \bigcup_i A_i^O$: finite set of alphabets.
- $T' \subseteq (Q \times A \times \Gamma) \times (Q \times \Gamma^*)$: set of transition relations. The transition relations are defined in fig 3.
- $G = \{F_1 \times Q_2 \times \ldots \times Q_n, \ Q_1 \times F_2 \times \ldots \times Q_n, \ \ldots, \ Q_1 \times \ldots \times Q_{n-1} \times F_n\}$: sets of final states.

Similarly, degeneralization can also be applied to generalized pushdown Büchi automaton.

Definition 9 (Degeneralization of pushdown GBA) Given a generalized pushdown Büchi automaton $B = (Q, q^0, A, \Gamma, z, T, F)$, where $F = \{F_1, \ldots, F_n\}$. There is an equivalent pushdown Büchi automaton $B' = (Q', q'^0, A, \Gamma, z, T', F')$, where

- $Q' = Q \times \{1, 2, \ldots, n\}$
- $q'^0 = (q^0, 1)$
- $F' = F_1 \times \{1\}$
- Γ' is defined as follows:
  Push:
  $((q_1,i),a,\gamma) \hookrightarrow ((q_2,j),a\gamma) \in T'$ iff
  $(q_1,a,\gamma) \hookrightarrow (q_2,a\gamma) \in T$ and
  $j = i$ if $q_1 \notin F_i$, $\quad j = (i \bmod k) + 1$ if $q_1 \in F_i$
  Pop:
  $((q_1,i),a,\gamma) \hookrightarrow ((q_2,j),\epsilon) \in T'$ iff
  $(q_1,a,\gamma) \hookrightarrow (q_2,\epsilon) \in T$ and $a = \gamma$ and
  $j = i$ if $q_1 \notin F_i$, $\quad j = (i \bmod k) + 1$ if $q_1 \in F_i$

Additionally, the model checking problem for a component base system with an adaptor is the same as without an adaptor, while description of properties needs to take consideration of head of stack. The behavior mismatch then becomes slightly different.

Definition 10 (Behavior Mismatch with adaptor) Given the degeneralized synchronous composition of a set of composable software components with an adaptor, which is represented by a pushdown Büchi automaton, Property of behavior mismatch is to define an atomic proposition p_{bmis} and a labeling function

$$L((s,\gamma)) : \{p_{bmis} | s \in F and \gamma = z\}$$

and write in the followubg form of LTL formula:

$$\Box\Diamond p_{bmis}$$

where z is start symbol of stack.

3.2 Coordinator Guided Adaptor Generation

Even the behavior of a component based system with an adaptor and how to model checking for mismatches are known, we still need a way to generate an appropriate adaptor. In this framework, we want to apply model checking again but in the way to help adaptor generation. The basic idea is to build a over-

behaviored adaptor called "coordinator" then prune unwanted traces from it. A simplest way is not to refine coordinator but to find a trace that can satisfies all properties. We can image that if we check the negation of properties, the returned counterexample is then an execution trace that the system can execute without mismatches. In detail, we propose an approach to generate adaptor from a counterexample:

(1) Given a set of composable software components, build a special adaptor called "coordinator" that over-behave to the system. That is, take transitions as many as possible the system could interactive with an adaptor.

(2) Model checking the negation of mismatch properties to get a counterexample if any. The counterexample can be simply converted to a pushdown automaton as an adaptor of the software components.

To define a coordinator for given software components, recall that an adaptor is represented by a pushdown automaton that uses alphabets and stack symbols same as alphabets of all components. In the case of coordinator, it only needs one state plus all possible transitions so that coordinator can send any alphabet on stack head if a component may receive it; coordinator can receive any alphabet and add it to stack head if a component may send it.

Definition 11 (Coordinator) Given a set of composable interface Büchi automata $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, F_i)$, $i \in [1, n]$. A coordinaot is a pushdown automaton

$$C = (Q_C, q_C^0, A_C, \Gamma, z, T)$$

where

- $Q_C = \{q_C^0\}$: finite set of states has only the initial state.
- q_C^0: initial state.
- $A_C = \bigcup_i A_i^I = \bigcup_i A_i^O$: finite set of alphabets.
- $\Gamma = A_C \cup \{z\} \cup \{\epsilon\}$: finite set of stack symbols.
- z: stack start symbol representing bottom of stack.
- $T = (Q_C \times A_C \times \Gamma) \times (Q_C \times \Gamma^*)$: set of transition relations.
  T is defined as follows:
  $< q_C^0, a, \gamma > \hookrightarrow < q_C^0, a\gamma >$: push
  $< q_C^0, a, \gamma > \hookrightarrow < q_C^0, \epsilon >$: pop
  where $a \in A_D$, $\gamma \in \Gamma$

Note that transitions of coordinator have full combinations of all alphabets so that coordinator can cover every possible communication of given software components.

We give the unformal algorithm of converting an counterexample to an adaptor as follows:

(1) Given a counter example C in the form of infinite sequence of configurations vw^ω, where $v = c_0 \dots c_i$ and $w = c_{i+1} \dots c_n$. A configuration of a pushdown system is in the form: (p, w) where $p \in Q$ is a state of the pushdown system and $w \in \Gamma^*$ is a finite word of stack symbol indicates the stack content.

(2) Take every p_i in the trace as the states of adaptor

(3) For every pair of configurations $c_j = (p_j, w_j)$ and $c_{j+1} = (p_{j+q}, w_{j+1})$, generate a transition between p_j and p_{j+1} by comparing w_j and w_{j+1}. Note that when $j = n$, $j + 1 = i + 1$.
   (a) if $|w_j| - |w_{j+1}| = 1$, generate a pop transition $< p_j, a, a > \hookrightarrow < q_{j+1}, \epsilon >$, where $a = head(w_j)$.
   (b) if $|w_j| - |w_{j+1}| = -1$, generate a push transition $< p_j, a, \gamma > \hookrightarrow < q_{j+1}, a\gamma >$, where $a = head(w_j)$ and $\gamma \in \Gamma$.

(4) for all sates $\{p_1, p_2, \dots, p_n\}$, apply a projection function: $proj(p) = (q_1, \dots, q_n)$, where $p = ((q_1, \dots, q_n, q_D, k)$. Merge all different states that have same projection as well as their transitions.

Then we can have an adaptor that has a set of states after merged with a set of generated transition rules.

4. Example

This section gives a simple example in detail to demonstrate the application of the framework. Consider a two-components system has behaviors draw in fig 4, we can easily figure out that the two components are compatible but have behvior mismatch, or specifically, reordering mismatch. We will now using the behavior mismatch property for the following demonstration.

The synchronous composition of the two components results in only one state that has no transition going out. It is obviously that behavior mismatch exists and we need ad adaptor to solve it. Because the trivil situation in this example, we omit the mismatch detection step.

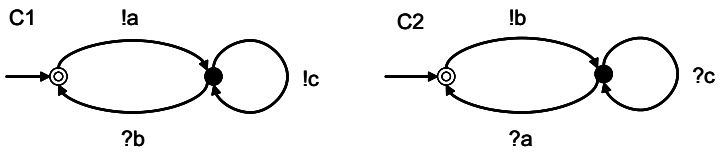

Fig. 4 An explanatory example

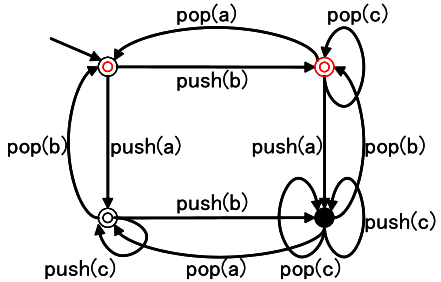

Fig. 5 explanatory example composition with coordinator

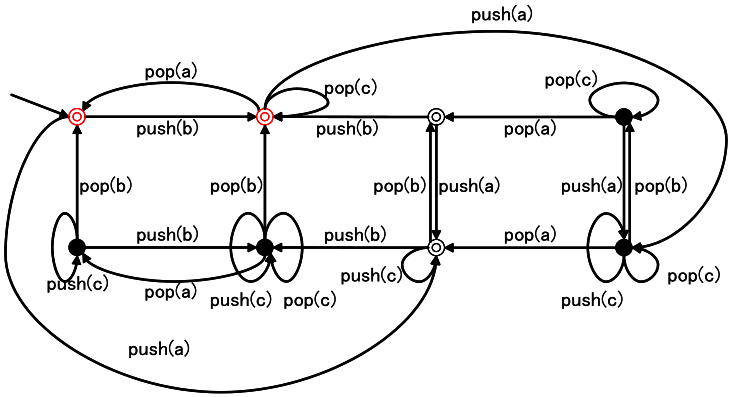

Fig. 6 explanatory example degeneralization with coordinator

To generate an adaptor, first we build a coordinator for this system, which only has one state and all push/pop transitions for all alphabets a, b, and c, we can compute the synchronous composition of the two component with coordinator to get a generalized pushdown Büchi automaton as shown in fig 5. After degeneralization, we get a pushdown Büchi automaton in fig 6.

Now we want apply model check of pushdown system with the neagtion of behavior mismatch property. The system behavior is then convert to a model of pushdown system that has no input alphabet. After checking the negation of behavior mismatch property and a counterexample returned. An adaptor for this example is generated as shown in fig 7.

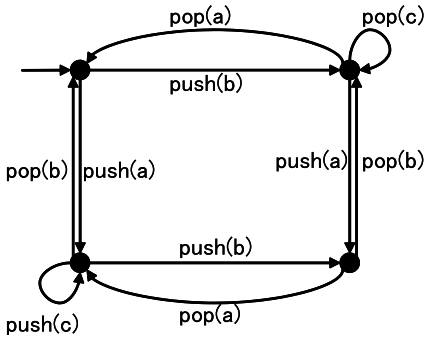

Fig. 7 explanatory example adaptor

5. Related Work

In the field of component adaptation, there are several researches that proposed solutions for the component mismatch problem. In general, researches of component adaptation can be categorized by using what behavior interfaces: for

example, automata are used by D. M. Yellin, et al[2]; process algebra modeling is used by A. Bracciali, et al[10]; Labeled transition system (LTS) is used by P. Inverardi, et al[8], C. Canal, et al[12] and M. Tivoli, et al[13]. Except[2] that applied adaptation only between two components, other solutions are based on direct composition of all behavior of components with deadlock elimination. Further more, adaptor generation by direct composition has ordering problem by LTS modeling. This problem is only be solved by[12] by using Petri-net modeling when ordering is needed. Compare to this paper, we propose an framework that integrates component adaptation and model checking while other researches only consider model checking a parallel procedure for component-based software design. By using pushdown automaton to model behavior of an adaptor, the ordering problem is not a problem in our approach. Finally, by our knowledge, only work of J. Cubo, et al[14] concerned about model checking but they yet did not integrate model checking with adaptor generation.

6. Conclusion and Future Work

This paper gives an framework of component adaptation with LTL model checking. The LTL model checking should at least check the liveness property to find behavior mismatch. In our approach software components are modeled by interface Büchi automata which capture the input/output protocol characteristics and continuous executing behavior of components such as web services. An adaptor is modeled by pushdown automaton which has a stack to remember received messages for sending out later. The LTL model checking of pushdown automata requires other algorithm than regular LTL model checking and we apply MOPED for model checking pushdown system. We also demonstrated our approach with an example and the result showed well.

Though our approach has the ability of automated adaptor generation, the tool is still under development. The tool will be capable of reading behavior interfaces and specified properties of a set of components, then do the detection of component mismatches and adaptor generation if needed.

References

1) Moshe Y. Vardi, and Pierre Wolper: “An automata-theoretic approach to automatic program verification,” Proc. First IEEE Symp. on Logic in Computer Science, 1986, pp. 322-331
2) Daniel M. Yellin and Robert E. Strom: “Protocol Specifications and Component Adaptors”, ACM Transactions on Programming Languages and Systems, Vol.19, No.2, pp. 292-333, 1997.
3) Javier Esparza, David Hansel, Peter Rossmanith, Stefan Schwoon: “Efficient Algorithms for Model Checking Pushdown Systems,” CAV 2000: 232-247.
4) Luca de Alfaro and Thomas A. Henzinger: “Interface Automata”, Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE’01), 2001.
5) Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, “Model Checking”, MIT Press, 2002.
6) Stefan Schwoon: “Model-Checking Pushdown Systems” , Ph.D. Dissertation, Technischen Universität München, 2002.
7) Gerard J. Holzmann: “The SPIN Model Checker: Primer and Reference Manual”. Addison Wesley, September, 2003
8) P. Inverardi and M. Tivoli: “Software Architecture for Correct Components Assembly,” Formal Methods for Software Architectures, pp. 92-121, 2003.
9) Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, Massimo Tivoli: “Towards an Engineering Approach to Component Adaptation,” Architecting Systems with Trustworthy Components 2004: 193-215
10) Andra Bracciali, Antonio Brogi and Carlos Canal: “A formal approach to component adaptation”, Journal of System and Software, Special Issue on Automated Component-Based Software Engineering, 2004.
11) Carlos Canal, Juan Manuel Murillo, Pascal Poizat: “Software Adaptation,” L’OBJET 12(1): 9-31 (2006)
12) Carlos Canal, Pascal Poizat, Gwen Salaün: “Model-Based Adaptation of Behavioral Mismatching Components,” IEEE Trans. Software Eng. 34(4): 546-563 (2008)
13) Massimo Tivoli, Paola Inverardi: “Failure-free coordinators synthesis for component-based architectures,” Science of Computer Programming 71(3): 181-212 (2008)
14) Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel, Pascal Poizat: “A Model-Based Approach to the Verification and Adaptation of WF/.NET Components,” Electr. Notes Theor. Comput. Sci. 215: 39-55 (2008)