Automatic Adaptor Generation for Behavioral Mismatches Using Pushdown Model Checking

by

Hsin-Hung Lin

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Associate Professor Toshiaki Aoki
Professor Takuya Katayama

School of Information Science
Japan Advanced Institute of Science and Technology

September, 2011

Abstract

Reuse of existing software components as well as web services by composition can save time and cost in application development. However, composition with existing components always encounters errors called mismatches. A promising solution for mismatches is adaptation which introduces a special component that controls interactions of components by exclusively synchronizing with components. Current approaches of adaptation focus on behavioral mismatches using automata modeling. With behavior interfaces of components and specified adaptor contracts designed by developers, adaptors can be automatically generated. Current approaches work well but still have some disadvantages. First, behavioral mismatches of behavioral interfaces may result in the need of an adaptor having non-regular behavior while in current approaches expressing non-regular behavior is not possible. Second, adaptation contracts are manually designed with limited tool support. This leads to difficulties especially in dealing with large scale systems or components need to be dynamically composed such as services in mobile devices.

This thesis analyzes the two disadvantages of current approaches of adaptation and introduces a novel approach of adaptor generation. The approach uses pushdown systems model as protocols of adaptors so that non-regular behavior is able to be expressed. In order to compute and generate adaptors with this new modeling, pushdown model checking is applied to generate suitable traces for further generating adaptors. The idea of generating suitable traces is letting model checking algorithms searching for the negation of necessary properties and generating counterexamples satisfying the properties. The adaptation is called "Coordinator Guided Adaptor Generation" which includes building a over-behavioral system with a special adaptor called Coordinator and generating adaptors from counterexamples returned from pushdown model checking. This way of using pushdown model checking can leave the design of adaptation contracts to algorithms of model checking and therefore achieves fully automatic adaptor generation. Additionally, in order to automatically capture necessary properties, behavior interfaces of components are remodeled with constraints to force the revelation of implicit information. Properties for behavioral-mismatch-free and unbounded messages are especially addressed in this thesis.

The approach is evaluated by experiments on a web service system having ordering behavioral mismatches. A non-regular behavioral adaptor was successfully and automatically generated. The generated adaptor was further implemented in BPEL processes to confirm the feasibility of the approach. The experiments shows the approach successfully performs automatic adaptor generation with non-regular behavior and the modeling using pushdown system can be easily implemented in BPEL processes as an example of platform of software development. For the generality of the approach, some technical issues are discussed. Solutions for signature mismatches and branchings are provided so that the approach is also applicable on general cases of adaptation.

Acknowledgments

First of all, I would like to express my sincere gratitude to my advisors Associate Professor Toshiaki Aoki and Professor Takuya Katayama of Japan Advanced Institute of Science and Technology. Professor Takuya Katayama guided my research direction on this work and Associate Professor Toshiaki Aoki helped me on completing this work especially on writing and presentation skills. They are also kindly provided constant encouragement as well as financial supports during this work.

I would also like to express my sincere gratitude to the members of examination committee of my doctoral dissertation, Professor Motoshi Saeki, Professor Tomoji Kishi, Professor Koichiro Ochimizu, and Professor Kokichi Futatsugi. Their precious comments helped me a lot on revising the final version of my dissertation. I would like to especially thank Professor Kokichi Futatsugi for his precious comments in my presentations in FMSD seminars. These comments greatly helped me on completeness of this work as well as improvements of my presentation skill.

I would also like to express my sincere gratitude to Ms. Sakurai and Ms. Morita as well as staffs of Secretarial Service Section. They helped me a lot on paper works for funding, traveling expenses, visa problem, and other issues of life in JAIST. Without their kindly support, I would not be able to complete this work.

I would also like to express my sincere gratitude the colleagues of Katayama Lab. and Aoki Lab for spending precious time on discussions and helping me with technical problems on machines.

There are some people I would like to express my sincere gratitude though I never meet them in person. First, I would like to express my thanks to one of the reviewers assigned to review my submission to FOCLASA'08. He/She gave me valuable comments and guidances for this work especially on selected references. I would also like to express my thanks to Assistant Professor Jonathan Sprinkle of University of Arizona. He kindly helped me solving problems on publishing my paper in ECBS 2011.

Finally, I would like to express my sincere gratitude to my parents for supporting me during this work with great patience and love. I would also like to express my sincere gratitude to my wife, Xuemei. I would never complete this work without her understanding and tolerance. Also, I would like to express my sincere gratitude to my two children, Paul and Olivia. Their innocent smiles and sleeping faces are the best encouragements to me.

Contents

List of Figures

Chapter 1

Introduction

1.1 Mismatches

1.1.1 Composition of Software Components

Developing software by reusing existing software components has became a common sense in software engineering. The most ideal and direct way of reusing software components is software composition. Since software components are basically wrapped within their interface by mature techniques from object-orient technologies, composition of software components can be analyzed and computed just by looking into their interfaces. However, software composition is not as simple as it seems because there are always errors encountered in composition especially when reusing existing software components. When building softwares from nothing, we may design all involved software components and ensure they are going to cooperate perfectly as one system. On the contract, building softwares using existing software components makes the scenario more complicate. Some of existing components may be developed with different considerations or under different design techniques or even specified by different interface definition languages. These differences surely result in errors in composition. Since developers usually choose components with confidences that the chosen components have desired functionalities and are worth to be composed for building new softwares, the errors in composition are not real errors but mismatches needed to be solved.

1.1.2 Levels of Mismatches

Mismatches are caused by gaps between interfaces of software components. How interfaces of software components are expressed is essential to clarify which mismatches are encountered. Furthermore, whether and how the encountered mismatches could be solved depend on interface descriptions of software components. According to survey work [1, 2], four levels of mismatches can be categorized:

- Technical mismatches: implementation related.
- Signature mismatches: different method names, parameter, etc.
- Behavioral/Protocol mismatches: deadlock in composition.
- Quality of Service mismatches: timeout setting for example.

- Semantic mismatches

Considering current technologies of interface description languages, signature and behavioral mismatches are taking most focuses. Under current interface descriptions languages, signatures of software components can be precisely described with behavioral descriptions using languages similar to automata model. The lowest level of mismatches, technical mismatches are related to implementation and is not a serious problem in recent software development paradigm using software components. Implementation details are encapsulated within a software component and only interface of the software component is revealed. For the higher levels of mismatches, we need description languages which can support QoS or Ontology contexts. This means we may need to introduce some specially designed description language for higher level mismatches and current interface description languages are not qualified for these requirements. Thus, current approaches for solving mismatches focus on signature and behavioral mismatches by given specifications of interfaces of software components. For the two levels of mismatches, the behavioral mismatches are especially interested since we can not detect behavioral mismatches by simply analyzing each behavioral interfaces but need to composed these interfaces and find out what is wrong and where is it.

1.1.3 Software Components and Web Services

When mentioned about software components, one may refer to Component Based Software Engineering(CBSE). In frameworks of CBSE, software components are encapsulated within interface specifications. Interface of a software component may also be considered its signature which includes methods and parameters to be invoked by other software components. For example, programing languages with object-oriented technologies implemented, such as JAVA and C++, can define interfaces of objects which are recognized as components. Some advanced interface definitions provide more expressiveness for interactions of components. For example, CORBA and COM provide more precise descriptions of interfaces which are also called protocols. In such frameworks, the interface descriptions about a component include not only signatures but also behavior of the component.

On the other hand, web services are considered as next generation of software components by introducing the standards of protocols of web services, such as WSDL, and BPEL, etc. The standards make web services more distributed and platform independent. Therefore, service oriented computing (SOC) not only inherits features and techniques from CBSE but also takes new challenges. Unlike software components, the framework of web services shown in Fig. 1.1 requires web application being developed by finding and composing web services registered on a repository. Technologies of SOC are therefore solutions for how to find and compose web services then create a web application as designed. In order to build a web application from services registered in the repository, developers first find services might be suitable for composing the designated web application according to interface descriptions of services published in the repository. Then tests are conducted to make sure the composed service behaves as designed.

It is known that in some aspects such as technologies relating to design and implementation, software components and web services are different. However, we argue that under the context of software reuse that developing softwares by composing existing software

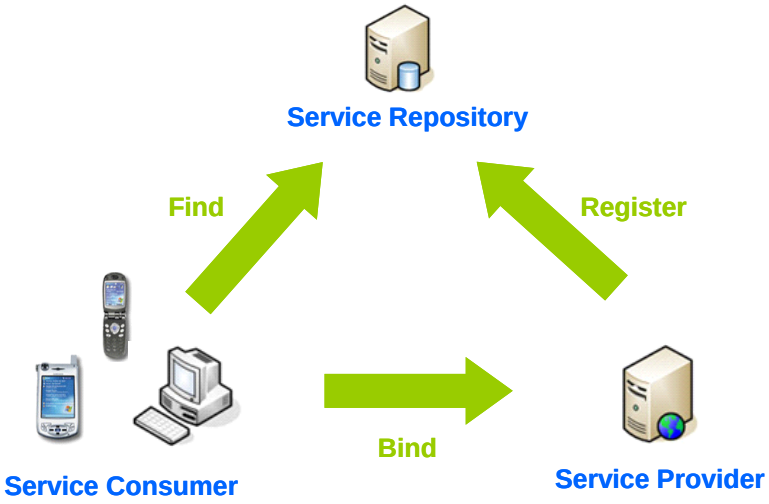
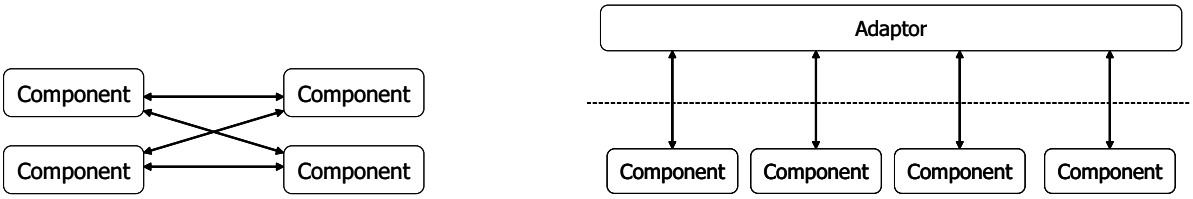


Figure 1.1: Framework of Web Services



(a) Direct composition

(b) Composition with adaptor

Figure 1.2: Adaptation

component, both software components and web services share the same vision: new softwares/web applications are developed by composing existing software components/web services. For example, commercial-off-the-shelf (COTS) in CBSE can be considered the same thing of publishing developed web services in a repository. Therefore, in this thesis, we consider software components and web services as same thing: components. In the rest of this thesis, the common term "component" is used to represent both software components and web services. Also, terms of "software components", "web services", or "services" should be recognized as just components if not addressed explicitly.

1.2 Adaptation

One of the most popular and promising solution for mismatches is adaptation. As shown in Fig. 1.2, adaptation introduces a special component called adaptor to coordinate communications of components. In an adapted system, all interactions among components are through the adaptor so that mismatches could be avoid without modifying given services. Thus, adaptation provides a non-intrusive way for composition of components with mismatches. For adaptation, components are treated as black boxes and only their interfaces are revealed.

Existing approaches for adaptation are, to our best knowledge, generally based on a

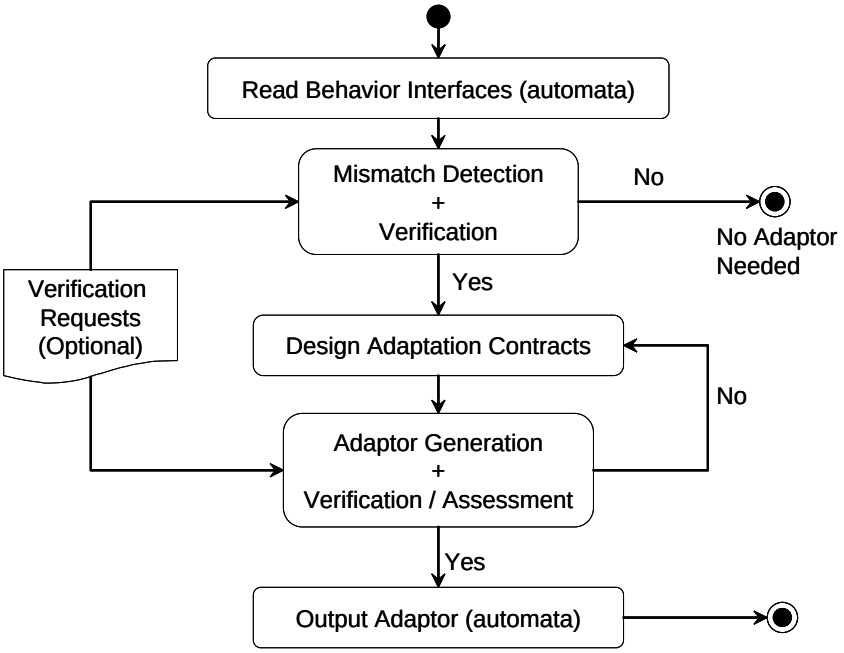


Figure 1.3: Adaptation: conventional framework

conventional framework of software adaptation [2] called model-based adaptation. The framework is designed to use *behavior interfaces* and *adaptation contracts* to perform automated adaptor generation. More specifically, behavior interfaces requires specifications of behavior interfaces of services as finite state automata represented in the form of labeled transition system (LTS), while adaptation contracts require specifications of mappings of messages to be interacted and the ordering (i.e., expected interactions coordinated by an adaptor) of these mappings represented in LTS. Fig. 1.3 shows a general flow of adaptation with verification considered [3]. In conventional approach, adaptation contracts are required to be designed in advance and adaptor is automatically generated based on specified adaptation contracts. It should be noticed that adaptation contracts are usually manually designed with interactive tool support. In the following of this thesis, when this framework will be called using terms "conventional framework of adaptation" or "conventional adaptation" for convenience when comparing to the approach in this thesis.

1.3 Problems

The conventional framework of adaptation works well and ways of computation are applied on this framework. However, there are still some problems with the conventional framework which motivates this work. This section presents the two main motivations of this work: (1) non-regular adaptation; (2) to overcome problems of adaptation contracts.

1.3.1 Non-regular Adaptation

The conventional framework of adaptation uses model of finite state machines with notations of LTS for representing behavior interfaces of components. The adaptation contracts are also specified using finite state machines. Since adaptation contracts describes how components should communicate/synchronize with each other so that mismatches will not be encountered. However, we argue that behavioral mismatches, especially ones caused by reordering of message delivery, may need adaptors with non-regular behavior to solve the mismatches. Thus, adaptors represented by finite state machines have nothing to do in this situation. To support adaptors with non-regular behavior, another model which can express non-regular behavior is needed. As a consequence, the computation of adaptation has to be changed to support adaptors represented by the new model.

1.3.2 Problems of Adaptation Contracts

It should be noticed and carefully recognized that in the conventional framework of adaptation, adaptation contracts are manually designed by developers with limited semi-automatic tool support [4, 5]. The purpose of adaptation contracts is for developers to specify how components should interact in the composed system so that mismatches will not be encountered while designated functionalities can be fulfilled. The natural that adaptation contracts have to be manually designed requires that developers know everything of the system including all provided behavioral interfaces, how interfaces interact. Though it is obvious that developers have to know well about behavioral interfaces of components, we argue that existing components makes the design of adaptation contracts more difficult since existing components are supposed not to be modified. When dealing with large scaled systems, the task of designing adaptation contracts becomes more complicated and is almost impossible to be manually done.

On the other hand, the recent trend of mobility of components, for example, web services on mobile devices, are required to be dynamically linked with other services. This scenario does not allow a step of designing adaptation contracts before composition. Thus, one may prefer that when a system needs an adaptor, the adaptor could be generated directly from given behavioral interfaces.

However, since the design of adaptation contracts means developers implant informations about the behavior of the system coordinated by adaptor. This information may not be originally included in behavioral interfaces of components. This information should be forced to be contained in behavioral interfaces in order to directly generate an adaptor. Therefore, in order to achieve the objective of directly generating adaptors from behavioral interfaces of components, a model of behavioral interfaces which are specially designated for adaptation, i.e., adaptor generation, of components is needed

1.4 Approach Overview

To achieve the two objectives described in Section 1.3, this work proposes a novel approach for adaptation of behavioral mismatching components. For the first objective, non-regular adaptation, this approach introduces pushdown systems model for representing behavior interface of an adaptor. The stack in a pushdown system gives the ability of describing non-regular behavior. Thus, for a system consists of components which needs an adaptor

with non-regular behavior, we can describe the adaptor in the approach. Since the model of an adaptor is different to conventional framework, computations such as synchronous composition should be re-constructed in the approach.

For the second objective of solving problem of adaptation contracts, the approach choose to skip the step of designing adaptation contracts but generate adaptors directly from behavior interfaces of components. In order to fill the lost information from developers through designing adaptation contracts, extra restrictions on behavior interfaces of components are introduced. These restrictions force the behavior interface of a component explicitly reveals information such as the start and end point of execution of a component, which is essential for generating an adaptor. Thus, the approach defines a modified model from Interface Automata called "Interface Automata for Adaptation" (IA4AD) as model of behavior interfaces of components. An IA4AD consists of basic information of behavior interface of a component represented as interface automata with constraints necessary for adaptation.

Steps of the approach is shown in Fig. 1.4 and details of these steps are described as follows:

- **Read Behavior Interfaces**:

  First behavior interfaces of components are specified by IA4ADs and used as input of the approach. This step not only get the behavior interfaces but also checks if restrictions of these IA4ADs are correctly specified. Furthermore, the behavior interfaces have to be tested if they satisfy the compatibility condition. Generally, in order to perform composition of components, components should meet the condition that all messages sent by one component must be received by another component. This means the components are composable or compatibility. By expressing behavior interfaces of components using model of IA4AD, we formally define the condition of compatibility of IA4ADs. Specified IA4ADs have to meet this condition before proceeding to further steps in the approach.

- **Detection of Behavior Mismatch**:

  If given components specified as IA4ADs pass the compatibility check, detection of behavior mismatches is performed to check if behavior mismatches exist. In the conventional framework of adaptation, behavioral mismatches are defined as the existence of deadlock states in the synchronous composition of components. Basically, a deadlock state in an automaton, i.e., the synchronous composition of components, can be intuitively defined as a state which can not reach final states of the automaton. Therefore, detecting behavioral mismatches can be considered as a search problem which checking the reachability to final states in an automaton.

  Since model checking techniques [6] are powerful and efficient search algorithms for transition systems, detecting behavioral mismatches by using model checking should be efficient than design our own searching algorithm. To apply model checking techniques, we need two inputs: one is the kripke structure of the automaton M and Linear-Time Temporal Logic (LTL) property representing reachability to final states. The kripke structure M can be obtained from the system behavior, i.e., the synchronous composition of components. Since behavior interfaces of components are represented by IA4ADs, the synchronous composition of IA4ADs is also an

IA4AD, which can be converted to the kripke structure M for model checking. The second input, a LTL formula which represent deadlock free for the system behavior computed from synchronous composition of components is simple. We may consider that deadlock free is a property that all traces of the system reach the final states. Thus, the LTL formula can be written as $\Diamond = p_{accept}$ where p_{accept} represents the acceptance condition of the system behavior, i.e., the current state the final state of the system. This property is called *Behavior Mismatch Free* in the approach.

Practically, we use SPIN [7] model checker on detection of behavior mismatch in the approach. Behavior interfaces of components represented in IA4ADs can be encoded in Promela with one synchronous message queue for communication. Then SPIN can do both synchronous composition and model checking for us. The final state of the system behavior is then the state where current states of all components are their final states. If the system behavior passes the check, there will be no counterexample returned. This means the system of components works well by themselves and no adaptation is needed. Otherwise, there will be a returned counterexample showing that adaptation is needed and we may proceed to the step of adaptor generation.

- **Adaptor Generation**:

  Adaptor generation in our approach is called *coordinator guided adaptor generation*. The idea of adaptor generation in the approach is model checking for negation of specified property to get a counterexample which is a trace of the system that satisfies specified property. Then the counterexample is used to build the behavior interface of an adaptor. In order to realize this idea, we introduce a special adaptor called *Coordinator*, i.e., a pushdown system, to synchronously compose with behavior interfaces of components, i.e., IA4ADs. *Coordinator* is expected to be an over-behavioral adaptor so that synchronous composition with Coordinator is also an over-behavioral adapted system. Thus, as long as we specify proper property for the desired adaptor, model checking for the negation of the property can give us a counterexample which is considered the behavior of the desired adaptor. Then an adaptor is generated according to the counterexample. In the approach, we introduce several properties to form the *proper* property including *Behavior Mismatch Free* used in detection of behavior mismatch. For generating correct non-regular adaptor, we also introduce the property of *Unbounded Messages* which is essential for non-regular behavior.

1.5 Outline of This Thesis

The structure of this paper is as follows. Chapter 2 gives some backgrounds of service adaptation and model checking, etc. Chapter 3 gives formal definitions of models of adaptors and components. Considerations of definitions are explained using a motivational example. Chapter 4 gives details of detection of behavior mismatch in the approach. Chapter 5 gives details of adaptor generation in the approach. Chapter 6 gives details of a tool implemented for the approach. Chapter 7 demonstrates experiments and results for selected problems of adaptation. Chapter 8 gives the evaluation to the approach based on results of experiments. Chapter 9 describes related work and compares to the approach

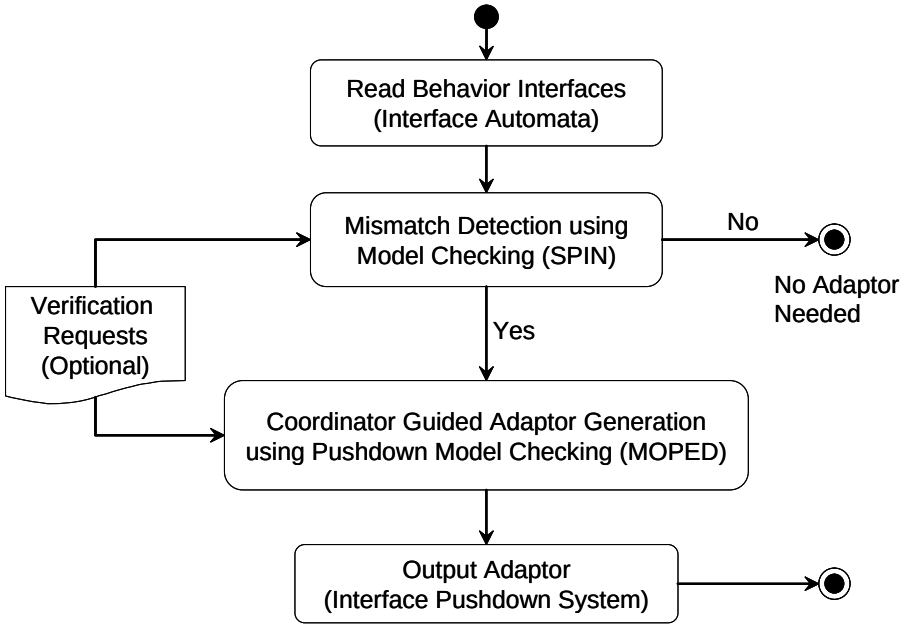


Figure 1.4: Service adaptation by pushdown model checking

proposed in this work. Chapter 10 summarizes this work by giving conclusions and future directions.

Chapter 2

Preliminaries

This chapter gives explanations and definitions of some backgrounds in this work. First we introduce software adaptation and then give definitions of Interface Automata, Pushdown Systems, etc. as theoretical base of this work.

2.1 Mismatches

Basically, mismatches between components are caused by gaps between interfaces of components, including existing and newly designed ones. Therefore, mismatches can be classified based on models of interfaces of components. According to S. Becker, et al. [8], interface models can be hierarchically classified from lowest to highest levels as:

- Signature List Based Interface
- Protocol Enhanced Interface
- Quality of Service Enhanced Interface

Signature list based interface model are interfaces like JAVA interfaces. Protocol enhanced interfaces are usually specified by finite automata model to reveal behavior of components such as BPEL for web services. Based on these interface models, mismatches between interfaces are classified as follows [1, 2]:

- Technical mismatches
- Signature mismatches
- Behavioral/Protocol mismatches
- Quality of Service mismatches
- Semantic mismatches

Technical mismatches are caused by gaps in implementation and are the lowest level of mismatches. Signature mismatches are caused by differences of signatures, i.e., method names, parameters, etc. Behavioral mismatches are caused by deadlocks in the composed system of components. QoS mismatches are caused by differences on non-functional descriptions such as timeout setting for example. Semantic mismatches are relating to understanding of functionalities, symbols, or keywords in description.

2.2 Software/Service Adaptation

Generally, service adaptation is considered evolved from software adaptation with extensions for dealing with web services such as standards of protocols. Therefore, basic techniques for service adaptation are common to software adaptation. We may call both service and software adaptation just adaptation for convenience. Also, the terms *software components* and *services* are considered the same things in this section for simplicity. This section introduces the conventional framework of adaptation based on the approach proposed by Canal et al. [2].

2.2.1 Behavioral Interfaces

Adaptation is a promising solution for signature and behavioral mismatchings. In a conventional framework of adaptation [2, 9, 10], the basic two elements for performing adaptation for software components having signature and/or behavioral mismatches are behavioral interfaces and adaptation contracts. Behavioral interfaces can be described by any IDL (Interface Description Language) and then are abstracted into Labeled Transition Systems (LTSs). A LTS is a kind of finite state machines and its definition is shown in Def. 1.

Definition 1 (LTS) *A labeled transition system is a tuple* (A, S, I, F, T) *where*

A*: an finite set of alphabets.*

S*: an finite set of states.*

$I \in S$*: initial state.*

$F \subseteq S$*: final states.*

$T \subseteq S \times A \times S$*: transition function.*

For a set of given software components, synchronous production are computed to composed the components. In order to perform the interactions between input and output alphabets, special symbols are used: `!a` and `?a` represent sending and receiving of messages `a` respectively. Since there is no definition of special symbol in LTS, the notation $\bar{a}$ is used to defined oppositional special symbols: $\bar{!a} = ?a$ and $\bar{?a} = !a$. The synchronous product is is defined in Def. 2.

Definition 2 (Synchronous Product) *The synchronous product of n LTSs* $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in [1, n]$, *is the LTS* $C_1 \| \ldots \| C_n = (A, S, I, F, T)$ *such that*

$A = A_1 \cup \{\_\} \times \ldots \times A_n \cup \{\_\}$

$S = S_1 \times \ldots \times S_n$

$I = (I_1, \ldots, I_n)$

$F = F_1 \times \ldots \times F_n$

T *is defined using the following rule:*
$\forall (s_1, \ldots, s_n) \in S$, $\forall i, j \in [1, n]$, $i < j$, *such that* $\exists (s_i, a, s_i') \in T_i$, $\exists (s_j, \bar{a}, s_j') \in T_j$*:*
$(x_1, \ldots, x_n) \in S$ *and* $((s_1, \ldots, s_n), (l_1, \ldots, l_n), (x_1, \ldots, x_n)) \in T$, *where* $\forall k \in [1, n]$*:*

$l_k = a,\ x_k = s'_i\ if\ k = i.$
$l_k = \bar{a},\ x_k = s'_j\ if\ k = j.$
$l_k = \_,\ x_k = s_k\ otherwise.$

It is intuitive that the synchronous product is also a LTS. It should be noticed that in synchronous production, only interaction of sending and receiving of same alphabet can be synchronized and be generated as a transition in the product LTS. For a product LTS, it is intuitive to check if there exists any deadlock state, i.e., a non-final state having no outgoing transition. If any deadlock state is found, there are behavior mismatches in the system of software components. To demonstrate how the behavioral interfaces are used, a example contains two software components is introduced in Example 1.

Example 1 *Fig. 2.1 shows a system consists of two components:* Library *and* User. Library *provides search service of books. By receiving* `title` *or* `author`*, it returns* `list`*: a list of books found in search. On the other hand,* User *only gives* `keyword` *that does not specify clearly title or author of a book and receives the search result* `list`*. The behavioral interfaces of the two components represented by LTS is as follows:*

$LTS_{Library}$

$$
\begin{aligned}
A &= \{\ ?author,\ ?title,\ !list\ \} \\
S &= \{\ s_0,\ s_1,\ s_2\ \} \\
I &= \{\ s_0,\ \} \\
F &= \{\ s_0,\ \} \\
T &= \{\ (s_0, ?author, s_1),\ (s_1, !list, s_0),\ (s_0, ?title, s_2),\ (s_2, !list, s_0)\ \}
\end{aligned}
$$

LTS_{User}

$$
\begin{aligned}
A &= \{\ !keyword,\ ?list\ \} \\
S &= \{\ s_0,\ s_1,\ \} \\
I &= \{\ s_0,\ \} \\
F &= \{\ s_0,\ \} \\
T &= \{\ (s_0, !keyword, s_1),\ (s_1, ?list, s_0)\ \}
\end{aligned}
$$

The synchronous product of the two component is a LTS having only initial state with no outgoing transitions since neither `title` *or* `author` *can synchronize with* `keyword`*. Thus, the two components have signature mismatches.*

2.2.2 Adaptation Contracts

When mismatching are found exist, adaptor generation is the next step. In conventional framework, adaptation contracts are specified manually by developer in the form of *synchronous vector*. As defined in Def. 3, a synchronous vector is as tuple that shows exchange of message corresponding to each participating component. We may use just the term vector in convenience.

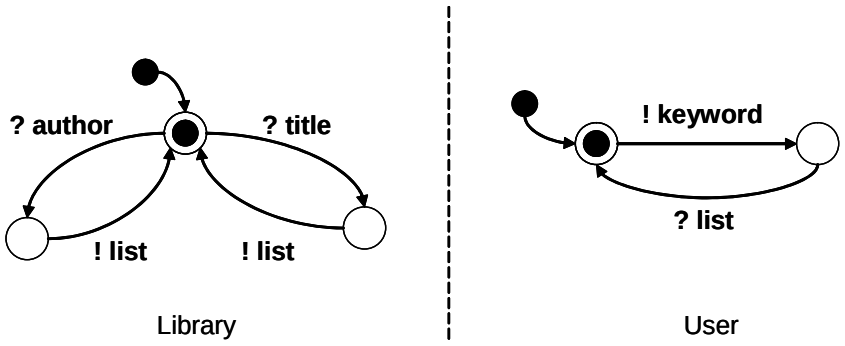


Figure 2.1: Example: Book Search

Definition 3 (Vector) *A synchronous vector for a set of components* $C_i = (A_i, S_i, I_i, F_i, T_i)$, $i \in [1, n]$, *is a tuple* $\langle e_1, \ldots, e_n \rangle$ *with* $e_i \in A_i \cup \{\_\}$, $\_$ *meaning that a component does not participate in a synchronization.*

Note that vectors should exhaustively list all messages that should be synchronized. This should include the simplest mapping: sending and receiving of same alphabet. Give all vectors are specified, adaptation contracts also need the ordering of executing these synchronizations. This is defined using vector LTS shown in Def. 4. Vectors and the corresponding vector LTS form the adaptation contracts for given system of software components. It is intuitive that the adaptation contracts is how the system should behave with no mismatchings encountered. Therefore, with behavior interfaces and adaptation contracts specified, adaptor is generated automatically by synchronizing LTSs of components with Vector LTS.

Definition 4 (Vector LTS) *An vector LTS for a set of vectors* V *is an LTS where labels are vectors.*

Since the computation of adaptor generation is automated, it is not necessary to show the detail of algorithms of adaptor generation. It should be noticed that algorithms of adaptor generation may be different by different approaches. However, all approaches based on this framework need specifications of behavior interfaces and adaptation contracts. Also, the generated adaptor is also represented as a LTS.

Example 2 *Example 2 already showed the two components* `Library` *and* `User` *have mismatches. Now we may design adaptation contracts for the two components. First, three vectors are specified as follows:*

$$V_{title} = \langle Library:\ ?title,\ User:\ !keyword \rangle$$
$$V_{author} = \langle Library:\ ?author,\ User:\ !keyword \rangle$$
$$V_{list} = \langle Library:\ !list,\ User:\ ?list \rangle$$

The first two vectors V_{title} *and* V_{author} *are mappings of messages* `title` *and* `author` *with* `keyword`*. The last vector* V_{list} *shows the mapping of* `list` *in both components. Note*

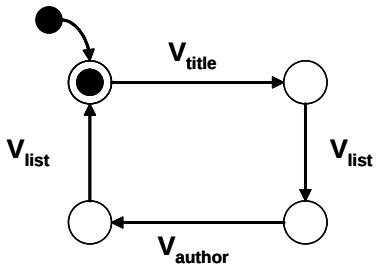


Figure 2.2: Book Search: vector LTS

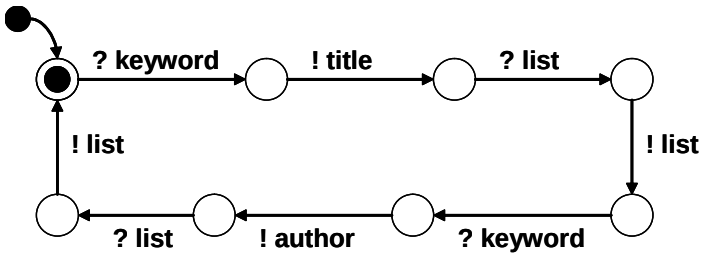


Figure 2.3: Book Search: adaptor

that V_{list} is required to be specified even the names in the mapping are the same. Next we design vector LTS to decide the ordering of the execution of the three vectors. Since `keyword` *should be used on any search of books, it is expected* `keyword` *be received and be used to generate list of books twice: one in title search and one in author search. Therefore, we design a vector LTS shown in Fig. 2.2. The generated adaptor is then shown in Fig. 2.3. It should be noticed that in the adaptor the special symbols are opposite to components since the adaptor are supposed to receive messages from components and sent back to corresponding components.*

Though the computation of adaptor generation is skipped in this example, the basic ideas of adaptation are clearly demonstrated. It is easy to see that the design of adaptation contracts are most difficult and important task when applying adaptation technique in real software components.

2.3 Interface Automata

Interface automata [11] defined in Def. 5 captures the input and output characteristic in behavior of software components. The explicit notation of input, output, and internal alphabets are especially convenient for expressing interactions between components. Note that an interface automaton does not have final states defined.

Definition 5 (Interface automata) *An interface automaton is a 6-tuple*

$$P = (V, v^0, A^I, A^O, A^H, \Delta)$$

where

V: *finite set of states.*

$v^0 \in Q$: *initial state.*

A^I: *finite set of input alphabets.*

A^O: *finite set of output alphabets.*

A^I: *finite set of internal alphabets.*

$\Delta \subseteq V \times A \times V$: *set of transition relations, where* $A = A^I \cup A^O \cup A^H$

It should be noticed that there is nearly no constraints in defining an Interface Automaton. Therefore, an Interface Automaton can be considered a general finite state machine with special notations. However, one condition called compatibility has to be satisfied when composing two components. As shown in Def. 6, compatibility defines the condition when two components are composable.

Definition 6 (Compatibility of interface automata) *Two interface automata* P *and* Q *are composable if*

$$A_P^H \cap A_Q = \emptyset \qquad A_P^I \cap A_Q^I = \emptyset$$
$$A_P^O \cap A_Q^O = \emptyset \qquad A_Q^H \cap A_P = \emptyset$$

Compatibility means that two components should have disjoint alphabets so that all shared alphabets are to be synchronized in composition and become internal alphabets. Therefore, let $shared(P,Q) = A_P \cap A_Q$, $shared(P,Q) = (A_P^I \cap A_Q^O) \cup (A_P^O \cap A_Q^I)$ for two composable components P and Q. Then the product of two Interface Automata is defined in Def. 7. Shared alphabets are added into internal alphabets and synchronization is made on corresponding transitions of the two components. Other non-shared alphabets, including input, output, and internal alphabets, are kept. Their corresponding transitions are also kept in th fashion of interleaving.

Definition 7 (Product of Interface Automata) *Given* P *and* Q *are composable interface automata, their composition is an interface automaton*

$$P \otimes Q = (V, v^0, A^I, A^O, A^H, \Delta)$$

where

$V = V_P \times V_Q$

$v^0 = (v_P^0, v_Q^0)$

$A^I = (A_P^I \cup A_Q^I) \setminus shared(P,Q)$, *where* $shared(P,Q) = A_P \cap A_Q$

$A^O = (A_P^O \cup A_Q^O) \setminus shared(P,Q)$

$A^H = A_P^H \cup A_Q^H \cup shared(P,Q)$

$\Delta \subseteq Q \times A \times Q$, *where* $A = A^I \cup A^O \cup A^H$ *is defined as follows:*
$\Delta = \{((v,u),a,(v',u))|(v,a,v') \in \Delta_P \wedge a \notin shared(P,Q) \wedge u \in V_Q\}$
$\quad \cup \{((v,u),a,(v,u'))|(u,a,u') \in \Delta_Q \wedge a \notin shared(P,Q) \wedge v \in V_P\}$
$\quad \cup \{((v,u),a,(v',u'))|(v,a,v') \in \Delta_P \wedge (u,a,u') \in \Delta_Q \wedge a \in shared(P,Q)\}$

In the definition of product of two Interface Automata, it should be noticed that the three situations of building transitions are not exhaustive. The situation that in a composite state when an alphabet a is in $shared(P,Q)$ and there is one component having corresponding transitions while the other component does not. A composite state in this situation is considered illegal because the synchronization fails when only one component tries to synchronize. In other word, product of two Interface Automata requires all shared alphabets being synchronized so that these alphabets become internal alphabets in the product Interface Automaton. The definition of illegal states is given in Def. 8.

Definition 8 (Illegal States) *The set of illegal states in the composition of two Interface Automata* $P \otimes Q$ *is defined as follows:* $Illegal(P,Q) = \{(v,u) \in V_P \times V_Q \mid \exists a \in shared(P,Q),\ ((a \in A_P^O(v) \wedge a \notin A_Q^I(u)) \vee (a \in A_Q^O(u) \wedge a \notin A_P^I(v)))\}$.

Though the existence of illegal states is not a problem for the product itself, the further product with another component is required that these illegal states are not included in any further product. Furthermore, the product of two Interface Automata is called composition if illegal states and corresponding transitions lead to illegal states are trimmed off. By using the product of Interface Automata, composition of software components is treated in a way of incremental manner.

Example 3 *A simple example from [11] is demonstrated to show how to compose Interface Automata. At first, there are two components* User *and* Comp *represented in Interface Automata shown in Fig. 2.4 and Fig. 2.5. Basically,* Comp *receives messages from* User *and send back responses. It should be noticed that besides the behavior of each component, the input and output alphabets are explicitly specified outside the behavior. This means that in an Interface Automaton the behavior inside and the input/output alphabets outside are basically independent. Therefore, component* User *is allowed to have input alphabet* `fail` *but no corresponding transition in its behavior. This directly causes an illegal state in the product* $User \otimes Comp$ *shown in Fig. 2.6. More specifically, the state marked with number 6 in Fig. 2.6 is illegal state in* $User \otimes Comp$ *because in this state, component* Comp *has a transition sending* `fail` *but component* User *does not have transition receiving* `fail` *to synchronize. Furthermore, by trimming off the state 6 and the transition connecting states 4 and 6, we get the composition of* User *and* Comp *which denoted as* $User \| Comp$. *From the experiences of this example, we may consider that by properly manipulating specifications in input/output alphabets and actual behavior, the system behavior obtained by computing product of components represented in Interface Automata can be maintained in control.*

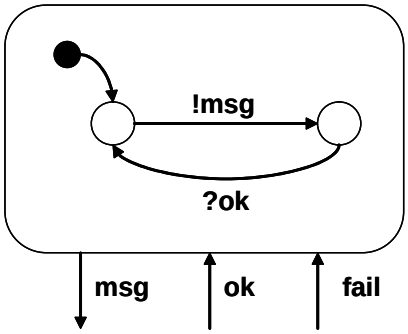


Figure 2.4: Interface Automaton: *User*

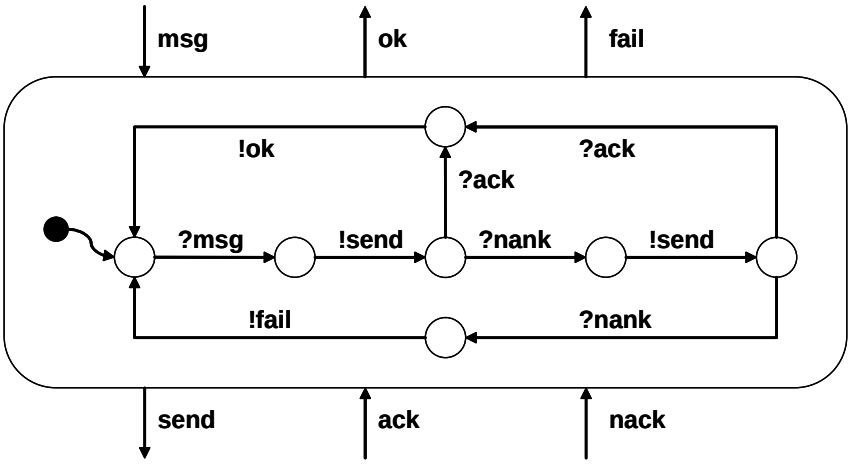


Figure 2.5: Interface Automaton: *Comp*

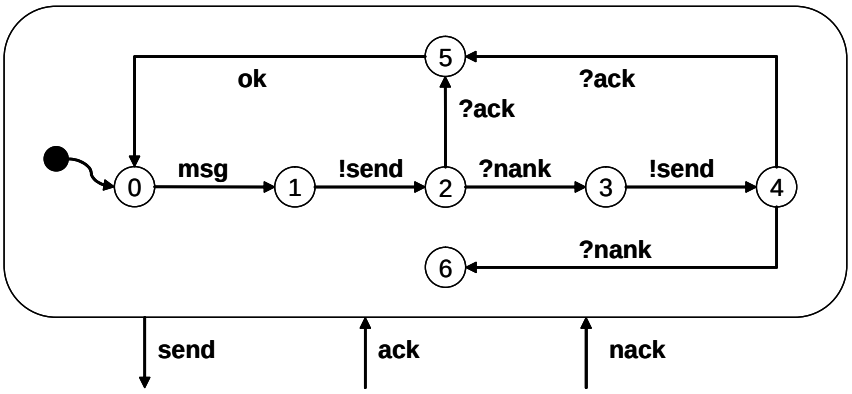


Figure 2.6: Product of *User* and *Comp*

2.4 LTL Model Checking

Model Checking [6] is an technique for verification of hardware/software design. Generally, given an abstract model represented by a transition system M (e.q. Kripke structure or finite automata) with a set of atomic propositions AP and a labeling function $L : S \to 2^{AP}$ labels each state with a subset of AP, a model checking problem is to check a specification φ composed of atomic propositions and temporal logic operators/qantifiers such as CTL (Computation Tree Logic) or LTL (Linear-Time Temporal Logic) and find all states such that $M, s \models \varphi$, and also check whether initial states are included.

However, LTL model checking is especially interested in software verification. A LTL formula represents a set of infinite traces and for every LTL formula we can build an *Büchi* automaton that accepts these infinite traces. Thus an automata-theoretic approach for LTL model checking [12] is useful.

The semantics of LTL formulas is as follows: Given an infinite execution trace $\sigma = s_0 s_1 \ldots$ with a set of atomic propositions AP, a labeling function $L : S \to 2^{AP}$, and arbitrary LTL formulas φ, φ_1, φ_2:

- $\sigma \models p$ 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 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*. Therefore, a LTL model checking problem is defined as follows: Given a model M with a labeling function L, and an LTL formula φ, check if all traces of M satisfy φ and return a counterexample if a trace does not satisfy φ. Since one can build a *Büchi* automaton that accepts traces of a LTL formula, the algorithm of LTL model checking on model M includes following steps:

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.

Model checking techniques advances very fast and there are tools supporting software verification. One of the most popular model checker is SPIN [7] which provides C program like input language called Promela. Developers can build the model of concurrently communicating protocols coded in Promela and SPIN can construct the model M using efficient techniques such as BDD.

2.5 Model Checking Pushdown Systems

Pushdown systems are simplified pushdown automata that input alphabets are omitted and focus on the behaviors rather than languages. In general, pushdown systems are used for representing procedure programs. Pushdown systems are simpler then ordinary pushdown automata, while the behavior of pushdown systems perfectly capture program counter of sequential programs especially recursive programs. Therefore, recently model checking pushdown systems [13] became popular and kept gaining the spotlight in the field of software verification.

2.5.1 Pushdown Automata

A pushdown Automaton is a finite state automaton with an unbounded stack. Languages accepted by pushdown automata are context free languages. Languages accepted by pushdown automata can be defined by final states or empty stack. Definition of pushdown automata is shown in Def. 9

Definition 9 (Pushdown automata) *A pushdown automaton is a 7-tuple*

$$M = (Q, q^0, A, \Gamma, z_0, \Delta, F)$$

where

Q*:finite set of states*

$q^0 \in Q$*: initial state*

A*: finite set of alphabets*

Γ*: finite set of stack symbols*

$z_0 \in \Gamma$*: initial symbol of stack*

$\Delta \subseteq (Q \times (A \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$*: set of transitions*

$F \subseteq Q$*: finite set of final states.*

2.5.2 Pushdown Systems

The definition of pushdown systems is given in Def. 10. It should be noticed that in a transition rule $(p, \gamma) \hookrightarrow (p', w) \in \Delta$, w does not represent the stack contents but only the word that replaces the head stack symbol γ, i.e., the topmost symbol in the stack, after the transition is fired.

Definition 10 (Pushdown System) *A pushdown system is a tuple*

$$\mathcal{P} = (Q, q^0, \Gamma, \Delta, z_0)$$

where

Q*: finite set of states.*

$q^0 \in Q$: *initial state.*

Γ: *finite set of stack symbols.*

$z_0 \in \Gamma$ *initial stack symbol.*

$\Delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$ *finite set of transition rules.*

A configuration of $\mathcal{P}$ *is denoted by a pair* $\langle p, w \rangle$ *where* $p \in P$ *and* $w \in \Gamma^*$. *The set of all configurations of* $\mathcal{P}$ *is denoted by* $Conf(\mathcal{P})$.

2.5.3 MOPED Model Checker

Since the stack is unbounded so the set of stack contents as well as the set of configurations are infinite. In order to apply model checking algorithms, first a pushdown system has to be transformed into a finite automaton called $\mathcal{P}$-automaton. Then model checking techniques and algorithms can be applied on $\mathcal{P}$-automaton. Based on the theories and algorithms, a model checker called MOPED for model checking pushdown systems is developed [14]. MOPED accepts two kind of inputs: (1) a directed encoded pushdown system; (2) boolean programs. Boolean programs are abstracted programs that can be abstracted from programs or built by developers. When the input is a boolean program, MOPED reads the boolean program and generate a pushdown system to perform model checking. In the case using a directly encoded pushdown system as input, pushdown model checking in MOPED basically searches for reachability of states or head stack symbols. Therefore the set of atomic propositions includes all states and stack symbols. Thus, we can build LTL formulas using states and stack symbols and logical connectors to express properties to be checked by MOPED.

Now A simple example of sequential programs from work [14] is demonstrated to show the usage of MOPED with pushdown system input. The example is called "plotter" consists of a group of simplified programs shown in Fig. 2.7. Generally, this example demonstrates a group of programs controlling a plotter device which plots charts or graphs, etc. on papers. The flow graph of plotter programs is shown in Fig. 2.8. It should be noticed that `m()` is a recursive function which calls itself. Therefore, model of finite state machines can not express the recursion and model of pushdown systems is introduced since the stack in a pushdown system can express the unbounded levels of recursions. According to the flow graph shown in Fig. 2.8, we may build a pushdown system $\mathcal{P} = (Q, q^0, \Gamma, \Delta, z_0)$ for the plotter example as follows:

$$Q = \{q^0\}$$

$$\begin{aligned}\Gamma = \{\ & m_0,\ m_1,\ m_2,\ m_3,\ m_4,\ m_5,\ m_6,\ m_7,\ m_8,\\ & s_0,\ s_1,\ s_2,\ s_3,\ s_4,\ s_5,\\ & main_0,\ main_1,\\ & up_0,\ right_0,\ down_0\ \}\end{aligned}$$

$$\begin{aligned}\Delta = \{\ & (q_0, m_0) \hookrightarrow (q_0, m_3),\ (q_0, m_0) \hookrightarrow (q_0, m_7),\\ & (q_0, m_3) \hookrightarrow (q_0, s_0 m_4),\ (q_0, m_4) \hookrightarrow (q_0, right_0 m_5),\\ & (q_0, m_5) \hookrightarrow (q_0, m_1),\ (q_0, m_5) \hookrightarrow (q_0, m_6),\end{aligned}$$

```
 1  void m() {
 2      if (?) {
 3          s(); right();
 4          if (?) m();
 5      } else {
 6          up(); m(); down();
 7      }
 8  }
 9
10  void s() {
11      if (?) return;
12      up(); m(); down();
13  }
14
15 void up() {
16     return;
17 }
18
19 void right() {
20     return;
21 }
22
23 void down() {
24     return;
25 }
26
27  main {
28      s();
29  }
```

Figure 2.7: The plotter example: programs

$(q_0, m_6) \hookrightarrow (q_0, m_0 m_1),\ (q_0, m_7) \hookrightarrow (q_0, up_0 m_8),$
$(q_0, m_8) \hookrightarrow (q_0, m_0 m_2),\ (q_0, m_2) \hookrightarrow (q_0, down_0 m_1),$
$(q_0, m_1) \hookrightarrow (q_0, \epsilon),$
$(q_0, s_0) \hookrightarrow (q_0, s_2),\ (q_0, s_0) \hookrightarrow (q_0, s_3),$
$(q_0, s_2) \hookrightarrow (q_0, up_0 s_4),\ (q_0, s_3) \hookrightarrow (q_0, \epsilon),$
$(q_0, s_4) \hookrightarrow (q_0, m_0 s_5),\ (q_0, s_5) \hookrightarrow (q_0, down_0 s_1),$
$(q_0, s_1) \hookrightarrow (q_0, \epsilon),$
$(q_0, main_0) \hookrightarrow (q_0, s_0 main_1),\ (q_0, main_1) \hookrightarrow (q_0, \epsilon),$
$(q_0, up_0) \hookrightarrow (q_0, \epsilon), (q_0, down_0) \hookrightarrow (q_0, \epsilon), (q_0, right_0) \hookrightarrow (q_0, \epsilon)\ \}$

Note that stack symbols can be mapped with the index numbers to states in flow graph shown in Fig. 2.8. For example, $main_0$ is the state marked in number 0 in the flow

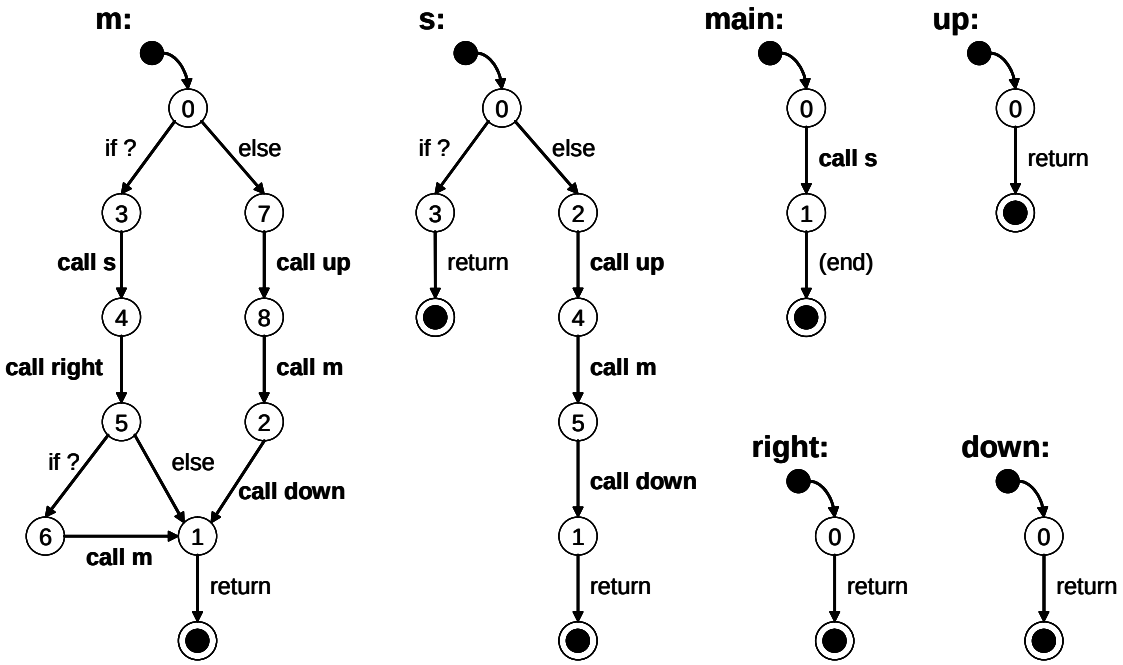


Figure 2.8: The plotter example: flow graph

graph of `main()`. The above pushdown system of plotter can be encoded as the input pushdown system of pushdown model checker MOPED shown in Fig. 2.9. The input pushdown system specifies first the initial configuration, i.e., initial state with stack start symbol, then the transition rules. It is easy to recognize that transition rules in Δ are intuitively encoded in the input pushdown system.

Now we can do some model checking. The execution of MOPED is in command line environment and the command to execute MOPED is in the form of

```
$ moped option filename property
```

The part `option` specifies whether printout trace of counterexample and other setting such as using specific algorithms. The part `filename` is the input pushdown system. The last part `property` is the property going to be checked. A property is represented as a LTL formula consists states and stack symbols which connected by logic connectors and temporal operators. For example, we may execute

```
$ mopde plot.pds '[](up0 -> (!down0 U up0 || right0))'
```

The filename `plot.pds` is the input file of this plotter example. In this check, the LTL property `[](up0 -> (!down0 U up0 || right0))` means for the plotter, an upwards movement is never immediately followed by a downward movement. Note that the atomic propositions in this LTL formula are stack symbols of the pushdown system. For example, the atomic proposition `up0` means the occurrence of stack symbol up_0 in the pushdown system, i.e., a configuration whose stack head is up_0 is reached. This LTL property for the plotter pushdown system is true so MOPED returns the result as follows:

```
 1  # Plotter example
 2
 3  (q <main0>)
 4
 5  # procedure m
 6  q <m0> --> q <m3>
 7  q <m0> --> q <m7>
 8  q <m3> --> q <s0 m4>
 9  q <m4> --> q <right0 m5>
10  q <m5> --> q <m1>
11  q <m5> --> q <m6>
12  q <m6> --> q <m0 m1>
13  q <m7> --> q <up0 m8>
14  q <m8> --> q <m0 m2>
15  q <m2> --> q <down0 m1>
16  q <m1> --> q <>
17
18  # procedure s
19  q <s0> --> q <s2>
20  q <s0> --> q <s3>
21  q <s2> --> q <up0 s4>
22  q <s3> --> q <>
23  q <s4> --> q <m0 s5>
24  q <s5> --> q <down0 s1>
25  q <s1> --> q <>
26
27  # procedure main
28  q <main0> --> q <s0 main1>
29  q <main1> --> q <>
30
31  # procedures up, down, right
32  q <up0> --> q <>
33  q <down0> --> q <>
34  q <right0> --> q <>
```

Figure 2.9: The plotter example: pushdown system

```
NO.

--- START ---
q <main0>
q <s0 main1>

--- LOOP ---
q <s2 main1>
q <up0 s4 main1>
q <s4 main1>
q <m0 s5 main1>
q <m3 s5 main1>
q <s0 m4 s5 main1>
```

Figure 2.10: The plotter example: counterexample

```
YES
```

Also, we may try another check for the termination of the plotter programs, i.e., whether stack symbol $main_1$ is reachable. We may write a LTL formula `<>main1` to see if the plotter programs always terminates and execute

```
$ mopde plot.pds '<>main1'
```

and get the negative answer `NO`. We may also want to see the counterexample so we can execute with the option `-t` to ask MOPED to output the trace of counterexample.

```
$ mopde -t plot.pds '<>main1'
```

We can get both the negative answer and a counterexample shown in Fig. 2.10. The counterexample is a trace with two parts: `START` and `LOOP`. By checking the counterexample, we may understand that the plotter programs may not terminate. It should be noticed that the configuration at the end of LOOP seems not consist with the first configuration of LOOP. To figure this out, we might confirm that the head symbol of the configuration right before LOOP is the same as the head symbol of the configuration at the end of LOOP. Then we may realize that the trace is the execution of the plotter programs so this trace indeed shows a never ending execution of plotter programs.

Chapter 3

Formal Definitions

This chapter introduces formal definitions of components and adaptors. In this work, existing models are modified for adaptation in the approach. Components are represented by *Interface Automata for Adaptation* (IA4AD) modified from Interface Automata. Adaptors are represented by *Interface Pushdown Systems* modified from pushdown systems. In this chapter, first a motivational example is given for demonstration of a typical non-regular adaptation problem. The motivational example will be used to explain details of formal definition including the modifications and the derivative definitions.

3.1 Motivational Example

During the survey of verification of services, we found an interesting example called "Fresh Market Update Service" in the work by X. Fu et al. [15]. The example is shown in Fig. 3.1 and descriptions of the example is in Example 4. We call this example *FMUS service* in short from now on. Detailed descriptions of SMUS service is shown in Example 4

Example 4 (FMUS service) *In the system shown in Fig. 3.1, there are three services in FMUS service.* Online Stock Broker *is supposed to send a list of* `RawData` *to* Research Department *for further analysis.* Investor *waits for the analyzed data* `Data` *from* Research Department*. It should be pay attention how* Research Department *processes the data analysis: once* Research Department *receives a* `RawData` *from* Online Stock Broker*, the raw data is processed and corresponding analyzed data* `Data` *is immediately sent out to* Investor*. When all* `RawData` *in the list are sent,* Online Stock Broker *will send* `EndOfData` *to* Research Department *to set the end of data transmission. Then* `Online Stock Broker` *will send* `Start` *to* Investor *as a signal telling* Investor *to start receiving analyzed data from* Research Department*. Similarly,* Research Department *sends* `Complete` *to inform* Investor *the finish of sending analyzed data. Finally,* Investor *confirms all analyzed data are received and sends* `Ack` *as acknowledgment. Therefore, current round of data processing is finished and the next round is ready to be started.*

In FMUS service, it should be noticed that each message is prefixed a special symbol "!" or "?" to indicate whether the transition represents a message sending or receiving. If we are going to synchronize the three services using synchronous composition, we encounter a problem that *Investor* enforces that `Start` has to be first received before any `Data` can

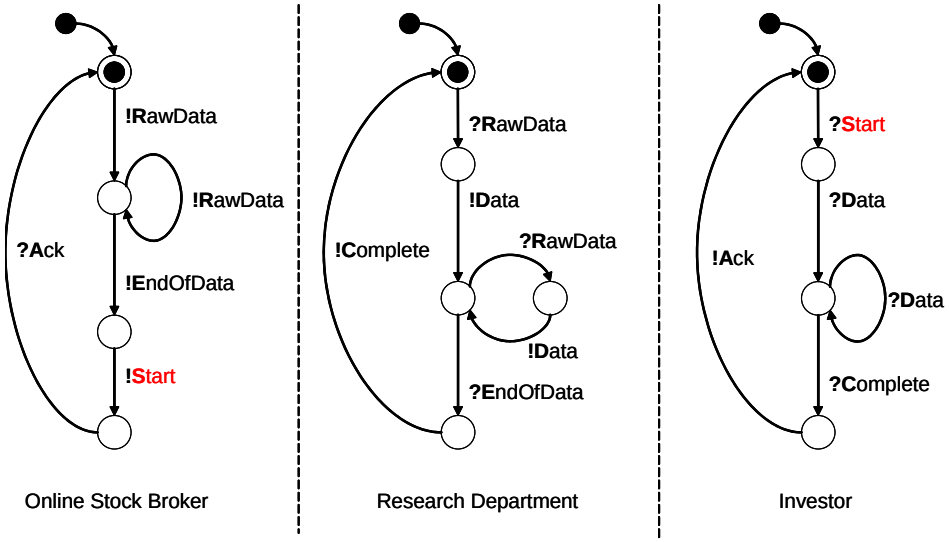


Figure 3.1: Fresh Market Update Service

be received. Thus, a behavioral mismatch caused by protocols of services is encountered in FMUS service.

We may try to apply the conventional framework of adaptation introduced in Section 2.2 to solve the behavioral mismatch in FMUS service. First three LTS corresponding to the three service is built and described in Example 5. Then we may design the adaptation contracts for the FMUS service. The adaptation contracts is constructed in Example 6. For the FMUS service, vectors are simple since there is no signature mismatch so we just write down pairs for all messages and make sure special symbols representing sending and receiving are correctly mapped in every pair. The vector LTS is designed to behave like the ω-regular language $(R^*ESD^*CA)^\omega$ where the capitals stand for the first alphabets of messages and represent corresponding vectors. For example, R represents $V_{RawData}$, and so on. By the LTSs and the adaptor contracts, the approach by C. Canal et al. can compute and generate an adaptor for us. The generated adaptor is shown in Fig. 3.3. It is easy to recognize the language expressed by the adaptor is $(?R\ !R\ ?R^*\ ?E\ ?S\ !S\ ?D\ !D\ \ (!R\ ?D\ !D)^*\ \ !E\ ?C\ !C\ ?A\ !A)^\omega$

Example 5 (LTSs in FMUS service) *Three LTSs can be built corresponding to the three services in FMUS service shown in Fig. 3.1:*

Online Stock Broker*:*
$P_1 = (A_1, S_1, I_1, F_1, T_1)$ *where*

$A_1 = \{\ !RawData,\ !EndOfData,\ !Start,\ ?Ack\ \}.$
$S_1 = \{\ s0,\ s1,\ s2,\ s3\ \}.$
$I_1 = s0.$
$F_1 = \{\ s0\ \}.$
$T_1 = \{\ (s0,\ !RawData,\ s1),\ (s1,\ !RawData,\ s1),\ (s1,\ !EndOfData,\ s2),$
$(s2,\ !Start,\ s3),\ (s3,\ ?Ack,\ s0)\ \}.$

Research Department:
$P_2 = (A_2, S_2, I_2, F_2, T_2)$ *where*

$A_2 = \{\ ?RawData,\ !Data,\ ?EndOfData,\ !Complete\ \}.$
$S_2 = \{\ s0,\ s1,\ s2,\ s3,\ s4\ \}.$
$I_2 = s0.$
$F_2 = \{\ s0\ \}.$
$T_2 = \{\ (s0,\ ?RawData,\ s1),\ (s1,\ !Data,\ s2),\ (s2,\ ?RawData,\ s3),$
$(s3,\ !Data,\ s2), (s2,\ ?EndOfData,\ s4),\ (s4,\ !Complete,\ s0)\ \}.$

Online Stock Broker:
$P_3 = (A_3, S_3, I_3, F_3, T_3)$ *where*

$A_1 = \{\ ?Start,\ ?Data,\ ?Complete,\ !Ack\ \}.$
$S_1 = \{\ s0,\ s1,\ s2,\ s3\ \}.$
$I_1 = s0.$
$F_1 = \{\ s0\ \}.$
$T_1 = \{\ (s0,\ ?Start,\ s1),\ (s1,\ ?Data,\ s2),\ (s2,\ ?Data,\ s2),$
$(s2,\ ?Complete,\ s3),\ (s3,\ !Ack,\ s0)\ \}.$

Example 6 *For the FMUS service shown in Fig. 3.1 with behavior interfaces of the three services represented in LTSs shown in Example 5, adaptation contracts are designed as follows:*

Vectors:

$V_{RawData} = \langle P_1 :\ !RawData,\ P_2 :\ ?RawData\rangle$
$V_{Data} = \langle P_2 :\ !Data,\ P_3 :\ ?Data\rangle$
$V_{EndOfData} = \langle P_1 :\ !EndOfData,\ P_2 :\ ?EndOfData\rangle$
$V_{Complete} = \langle P_2 :\ !Complete,\ P_3 :\ ?Complete\rangle$
$V_{Start} = \langle P_1 :\ !Start,\ P_3 :\ ?Start\rangle$
$V_{Ack} = \langle P_3 :\ !Ack,\ P_1 :\ ?Ack\rangle$

Vector LTS:

See Fig. 3.2

Recall the problems of conventional framework of adaptation mentioned in Section 1.3, we now give more concrete discussions using the FMUS service example.

- **Non-regular Adaptation**:

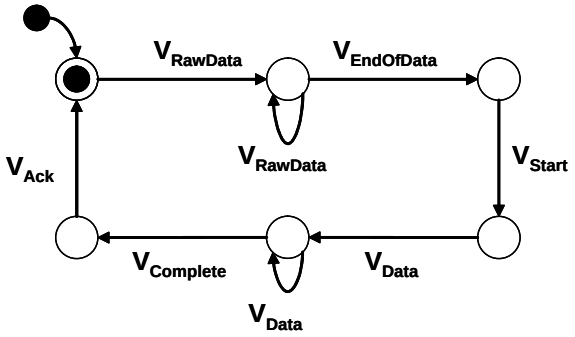


Figure 3.2: Vector LTS for FMUS service

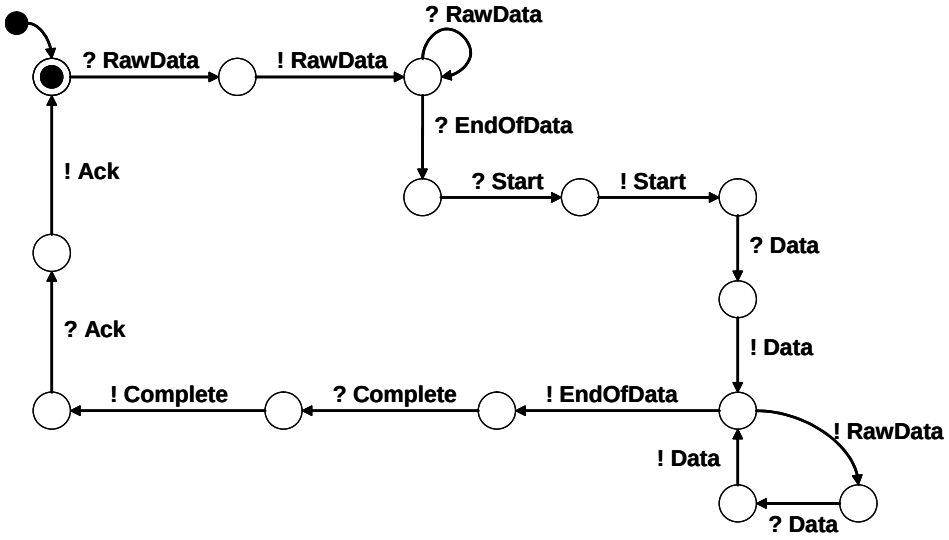


Figure 3.3: adaptor (LTS) for FMUS service

The generated adaptor shown in Fig. 3.3 seems quite appropriate and correctly reflects the structures of the three behavior interfaces of services. However, with carefully checking the behavior of the three services described in Fig. 3.1, the behavior of generated adaptor is not exactly the behavior the system is looking for. Note that in the description of functionalities of FMUS service in Section 3.1, *Research Department* does data analysis in the following fashion: for every `RawData` received, corresponding `Data` is generated and sent out to *Investor*. Therefore, numbers of `RawData` and `Data` must be the same so that all data in the list of raw data received from *Online Stock Broker* is analyzed and sent out to *Investor*. Thus, the FMUS service actually expects an adaptor which expresses non-regular ω-languages such as $(?R\ !R\ ?D)^n\ ?E\ !E\ ?S\ !S\ !D^n\ ?C\ !C\ ?A\ !A)^\omega,\ n > 1$. Note that the natural number n in the language is the key factor that makes the behavior of expected adaptor non-regular. We may call the adaptor with non-regular behavior the *expected adaptor* and the adaptor shown in Fig. 3.3 the *LTS adaptor* for convenience. The LTS adaptor expressing the regular ω-language $(?R\ !R\ ?R^*\ \ ?E\ ?S\ !S\ ?D\ !D\ (!R\ ?D\ !D)^*\ \ !E\ ?C\ !C\ ?A\ !A)^\omega$ does not maintain the numbers of `RawData` and `Data` in its behavior. This is the limitation of LTS and all other finite state machines. Thus, another model that supports non-regular behavior like the expected adaptor is necessary for solving the behavioral mismatch in FMUS service. This gives us another motivation in this work: select and use a model that expresses non-regular languages suitable for representing adaptors. We need to investigate elements that characterize the non-regular behavior in adaptors.

- **Problems of Adaptation Contracts**:

  Despite the computation of automated adaptor generation, the most valuable task in doing adaptation for the FMUS service is the design of adaptor contracts, especially the vector LTS shown in Fig. 3.2. It is reasonable to consider that the correctness of a generated adaptor mostly count on the design of adaptation contracts. However, designing adaptation contracts requires a thorough understanding for all given components. The understanding is required to include behavior and functionalities of each separate component and the synchronous composition of components. When dealing with large scale systems, manually designing adaptation contracts is nearly impossible especially in the case of solving reordering behavior mismatch like the FMUS service. If we can force the behavior interfaces of components reveals necessary consideration of adaptation contracts, especially behavior of an adaptor, we might be able to generate adaptors directly from behavior interfaces of components. Therefore adaptation contracts is no longer needed in adaptation and fully automatic adaptor generation is possible.

It should be noticed that above discussions are not independent issues. The model we use to represent an adaptor, the way we generate an adaptor, and the way we specify behavior interfaces of components are all coupled together.

3.2 Model of Components

As mentioned in Section 3.1, models of components and adaptors are closely coupled. Thus, selection of the model of components should consider about what model of adaptors

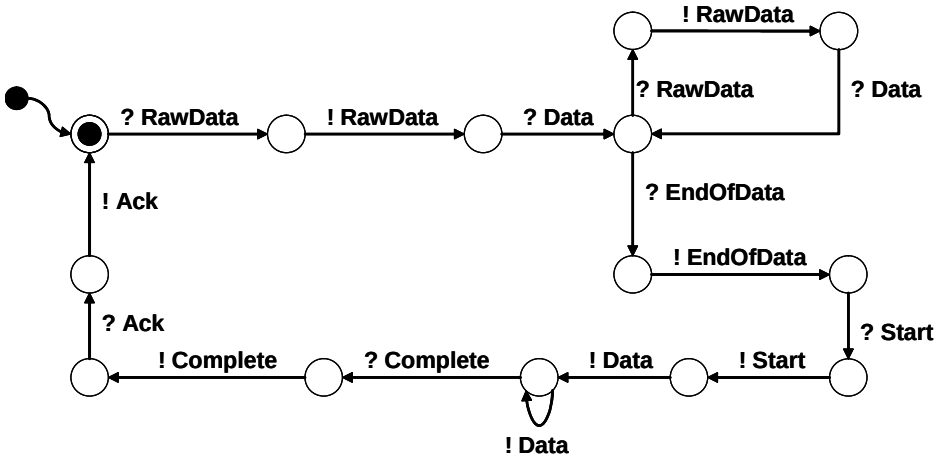


Figure 3.4: FMUS service: the expected adaptor

we want, and vice versa. For convenience of discussing considerations about selection of models of components and adaptors, we may give some assumptions of the model of adaptors and then discuss the model of components. Lets take a look again on the behavior of the expected adaptor $(?R\ !R\ ?D)^n\ ?E\ !E\ ?S\ !S\ \ !D^n\ ?C\ !C\ ?A\ !A)^\omega,\ n > 1$ mentioned in Section 3.1. Though we will define the model of adaptors as *Interface Pushdown Systems* in the Section 3.3 later, we may say here that the model suitable for expressing this behavior is *pushdown automata* model. Thus, we can still catch the ideas of the model of components in the approach and leave the detailed discussions about the model of adaptors in Section 3.3 later. Since the objective is to generate an adaptor which behaves like the expected adaptor, we try to build a pushdown automaton for expressing the behavior of the expected adaptor. We may assume the three services in FMUS services are defined in LTS following the conventional framework, then build a pushdown automaton shown in Fig. 3.4. For convenience, labels on transitions are demonstrated by sending and receiving of messages using special symbols "?" and "!" as prefix. The detailed definition of the pushdown automaton of the expected adaptor is shown in Example 7. This definition as an pushdown automaton can be considered as a first tempt of formalization non-regular adaptors using the expected adaptor as an example. The key part is how to encode message receptions and deliveries in an adaptor in to transitions involving operations of pushing an popping stack symbols. It is intuitive to connect receiving a message with pushing a symbol into the stack and similarly delivering a message with popping a symbol at the head of the stack. Therefore, in Example 7, the alphabets are not prefixed with special symbols and the sending and receiving of messages are defined as pushing and popping stack symbols.

Example 7 (Expected Adaptor of FMUS service) *The expected adaptor with behavior*

$$(?R\ !R\ ?D)^n\ ?E\ !E\ ?S\ !S\ !D^n\ ?C\ !C\ ?A\ !A)^\omega,\ n > 1$$

is defined as a pushdown automaton

$$D_{expected} = (Q, q^0, A, \Gamma, z_0, \Delta, F)$$

where

$Q = \{\ q0,\ q1,\ q2,\ q3,\ q4,\ q5,\ q6,\ q7,\ q8,\ q9,\ q10,\ q11,\ q12,\ q13\ \}$.

$q^0 = q0$.

$A = \{\ RawData,\ Data,\ EndOfData,\ Complete,\ Start,\ Ack\ \}$.

$\Gamma = A \cup \{\ z\ \}$.

$z \in \Gamma$ *is initial symbol of stack.*

$$\begin{aligned}\Delta = \{\ & (q0,\ RawData,\ \epsilon) \hookrightarrow (q1,\ RawData),\\ & (q1,\ RawData,\ RawData) \hookrightarrow (q2,\ \epsilon),\\ & (q2,\ Data,\ \epsilon) \hookrightarrow (q3,\ Data),\\ & (q3,\ RawData,\ \epsilon) \hookrightarrow (q4,\ RawData),\\ & (q5,\ RawData,\ RawData) \hookrightarrow (q5,\ \epsilon),\\ & (q5,\ Data,\ \epsilon) \hookrightarrow (q3,\ Data),\\ & (q3,\ EndOfData,\ \epsilon) \hookrightarrow (q6,\ EndOfData),\\ & (q6,\ EndOfData,\ EndOfData) \hookrightarrow (q7,\ \epsilon),\\ & (q7,\ Start,\ \epsilon) \hookrightarrow (q8,\ Start),\\ & (q8,\ Start,\ Start) \hookrightarrow (q9,\ \epsilon),\\ & (q9,\ Data,\ Data) \hookrightarrow (q10,\ \epsilon),\\ & (q10,\ Data,\ Data) \hookrightarrow (q10,\ \epsilon),\\ & (q10,\ Complete,\ \epsilon) \hookrightarrow (q11,\ Complete),\\ & (q11,\ Complete,\ Complete) \hookrightarrow (q12,\ \epsilon),\\ & (q12,\ Ack,\ \epsilon) \hookrightarrow (q13,\ Ack),\\ & (q13,\ Ack,\ Ack) \hookrightarrow (q0,\ \epsilon)\ \}\end{aligned}$$

$F = \{\ s0\ \}$.

However, using this way of definition may cause a serious issue: the defined pushdown model in Example 7 does not exactly express the behavior of the expected adaptor. This issue is related to *unbound messages* which are sending and receiving arbitrary multiple times. Comparing the behavior of the expected adaptor with the pushdown automaton we built, we may find that the arbitrary natural number n is defined as loop transitions in the pushdown automaton. Therefore, n is not explicitly expressed but implicitly constrained by the stack. Generally with the help of the stack, all pushed symbols are supposed being popped out later so that n should be promised. Unfortunately, the structure of the pushdown automaton that define the initial and the final states as same state leads to unexpected behavior such as

$$\begin{aligned}&((?R\ !R\ ?D)^3\ ?E\ !E\ ?S\ !S\ !D^2\ ?C\ !C\ ?A\ !A\\ &(?R\ !R\ ?D)^2\ ?E\ !E\ ?S\ !S\ !D^3\ ?C\ !C\ ?A\ !A))^\omega\end{aligned}$$

The numbers 3 and 2 shown in the above behavior should be carefully recognized. In this behavior, one `Data` is kept in the stack while the three services proceed to next execution. In the second execution, the `Data` kept in the first execution is popped out so that the stack is empty at the end of the second execution. This results that two executions of the system makes one "execution" of the above behavior. Furthermore, the above behavior is

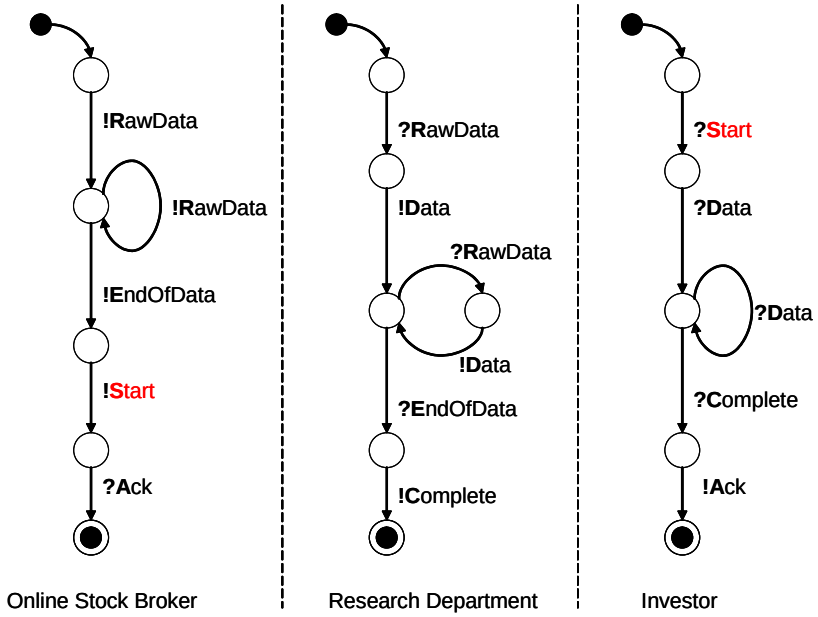


Figure 3.5: Sessional Fresh Market Update Service

only one of many variations that use multiple executions of the system as one "execution" in adaptors. This is not acceptable since in the system of FMUS service, all `RawData` and `Data` sent have to be consumed by their target services at the end of one execution. Though this is not explicitly specified, we should make the end of execution explicitly clear to avoid unexpected behavior like the above behavior.

Therefore, we demand that all components have to define their end of execution explicitly clear. The end of execution of a component means the point where the achievement of its functionalities is done. In this work, a component is not allowed to define its initial state and final state as same state. The initial and final states should be defined separately in different states so that the end of execution of a component can be easily recognized. According to this requirement, the behavior of the expected adaptor now becomes

$$(?R\ !R\ ?D)^n\ ?E\ !E\ ?S\ !S\ !D^n\ ?C\ !C\ ?A\ !A),\ n > 1$$

where the ω is removed. The three services in the FMUS service should also make clear their end of execution and are re-specified as shown in Fig. 3.5. We may call components having initial and final states distinctly without re-start from the final state *components as one session process*. It should be noted that though a component is defined having only one final state, the execution is looped in implementation. This means when the final state of a component is reached, the current state is expected to be shifted to the initial state for the next execution to be proceeded. Now we may call the FMUS service shown in Fig. 3.5 *sessional FMUS service* for references in the rest of this thesis. Similarly, we use the behavior of the *expected sessional adaptor* to distinguish from the first one with ω.

Now we start to give formal definitions of the model of components. As already mentioned in Section 1.4, the model of components in this work is modified from the definition of Interface Automata. The reason of using Interface Automata instead of using usual automata model is described as follows:

- Interface Automaton introduce the notations for input, output, and internal alphabets. This explicitly expresses the feature of communicating services/components. Compare to LTS model in conventional framework that only applies special symbols as prefixes, the use of input/output notations in Interface Automata makes it easier in defining interactions of services/components. We argue that the synchronous product in the conventional framework defined in Def. 2 is not easy to be understand because there are too many additional symbols used in the definition where not explicitly defined in definition of LTS model. On the other hand, the definition of synchronous product of Interface Automata defined in Def. 7 can be easily understand without additional explanation.
- Furthermore, Interface Automata provide definition of compatibility in Def. 6 that makes clear constraints of composing Interface Automata. Since automated adaptor generation is one of the main objectives in this work and there are some constraints needed for this objective, we may adopt the constraints in Interface Automata to fit the constraints needed in this work. Therefore, we argue that Interface Automata model is better than LTS model.

We call the model of components in this work "Interface Automata for Adaptation" which is written as IA4AD in short and defined in Def. 11. To keep the generality of Interface Automata, the definition keeps internal alphabets to allow internal transitions which does no communication. This also makes it intuitive and easy to apply on models expressed using process algebra. Furthermore, in some cases, there may be needs of expressing indeterministic behavior where using internal transitions may help.

Definition 11 (IA4AD) *An interface automaton for component is defined as*

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

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* $A = A^I \cup A^O \cup A^H$

$q^f \in Q_i$*: final state.*

An IA4AD has to satisfy the following conditions:

$q^0 \neq q^f$

$\nexists t \in \Delta,\ t = (q, a, q'),\ q, q' \in Q,\ q = q^f \vee q' = q^0$

$\forall a \in A.\ \ \exists t \in \Delta,\ t = (q, a, q'),\ q, q' \in Q$

Note that IA4AD has q^0 and q^f representing start and end of a component while in original Interface Automata there is only initial state but no final/accepting state. Recall the considerations mentioned in Section 3.1, we put the idea of *component as one session process* into the definition of model of components as constraints for adaptation. The constraint that the start and end state of a component are required to be different states to make it clear where a component starts and ends. Let $L(P)$ is the set of traces start from q^0 and end at q^f, we can define runs and acceptance runs of IA4AD in Def. 12 and Def. 13. According to the two definitions, the acceptance condition of an IA4AD is the reachability of its final state q^f.

Definition 12 (Runs of IA4AD) *A finite trace $\sigma = s_0 s_1 s_2 \dots s_k$ is a run of an IA4AD $P = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$ if $s_0 = q^0$ and $\forall i \in [1, k-1].\ \exists \delta \in \Delta,\ \delta = (s_i, a, s_{i+1})$, $a \in A$.*

Definition 13 (Accepting Runs of IA4AD) *A run $\sigma = s_0 s_1 s_2 \dots s_k$ of an IA4AD $P = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$ is an accepting run if $s_k = q^f$.*

Now we can define the three web services in the *sessional* FMUS service shown in Fig. 3.5. The three services of the sessional FMUS service are defined using IA4AD in Example 8. Note that prefixes in labels of transition relations are not necessary and only for convenience of recognition.

Example 8 (IA4ADs in FMUS service) *Three IA4ADs can be built corresponding to the three services in* sessional *FMUS service shown in Fig. 3.5:*

Online Stock Broker*:*
$P_1 = (\ Q_1,\ q_1^0,\ A_1^I,\ A_1^O,\ A_1^H,\ \Delta_1,\ q_1^f)$ *where*

- $Q_1 = \{\ q0_1,\ q1_1,\ q2_1,\ q3_1,\ q4_1\}$.
- $q_1^0 = q0_1$.
- $A_1^I = \{\ Ack\ \}$.
- $A_1^O = \{\ RawData,\ EndOfData,\ Start\ \}$.
- $A_1^H = \emptyset$.
- $q_1^f = q4_1$.
- $\Delta_1 = \{\ (q0_1,\ !RawData,\ q1_1),\ (q1_1,\ !RawData,\ q1_1),$
  $(q1_1,\ !EndOfData,\ q2_1),\ (q2_1,\ !Start,\ q3_1),$
  $(q3_1,\ ?Ack,\ q4_1)\ \}$.

Research Department*:*
$P_2 = (\ Q_2,\ q_2^0,\ A_2^I,\ A_2^O,\ A_2^H,\ \Delta_2,\ q_2^f)$ *where*

- $Q_2 = \{\ q0_2,\ q1_2,\ q2_2,\ q3_2,\ q4_2,\ q5_2\}$.
- $q_2^0 = q0_2$.
- $A_2^I = \{\ RawData,\ EndOfData\ \}$.

- $A_2^O = \{\ Data,\ Complete\ \}$.
- $A_2^H = \emptyset$.
- $q_2^f = s5_2$.
- $\Delta_2 = \{\ (q0_2,\ ?RawData,\ q1_2),\ (q1_2,\ !Data,\ q2_2),\ (q2_2,\ ?RawData,\ q3_2),$
  $(q3_2,\ !Data,\ q2_2), (q2_2,\ ?EndOfData,\ q4_2),\ (q4_2,\ !Complete,\ q5_2)\ \}$.

Online Stock Broker*:*
$P_3 = (\ Q_3,\ q_3^0,\ A_3^I,\ A_3^O,\ A_3^H,\ \Delta_3,\ q_3^f)$ *where*

- $Q_3 = \{\ q0_3,\ q1_3,\ q2_3,\ q3_3,\ q4_3\}$.
- $q_3^0 = q0_3$.
- $A_3^I = \{\ Start,\ Data,\ Complete\ \}$.
- $A_3^O = \{\ Ack\ \}$.
- $A_3^H = \emptyset$.
- $q_3^f = q4_1$.
- $\Delta_3 = \{\ (q0_3,\ ?Start,\ q1_3),\ (q1_3,\ ?Data,\ q2_3),\ (q2_3,\ ?Data,\ q2_3),$
  $(q2_3,\ ?Complete,\ q3_3),\ (q3_3,\ !Ack,\ q4_3)\ \}$.

Since the purpose in the approach is composition of components, we also need requirements on set of components. For a given set of components represented by IA4ADs, it is required that for every alphabet in the set of input/output alphabets of an IA4AD, there must be at least one transition whose label is the alphabet. This is to make sure that all input/output alphabets are supposed to be synchronized with transitions in other components. This is intuitive because when signatures of sending or receiving of a messages is specified in behavior interfaces of components, the message is expected to be synchronized in the composition with other components. Furthermore, this consistency between alphabets and transition labels is also essential for adaptation generation in our approach since the alphabets in an adaptor is supposed to be the union of all alphabets of all services. Thus, we need to define a condition for a set of IA4ADs to be composed. This is called *compatibility* in the approach. Recall that there is already the condition of compatibility defined in Def. 6 of Section 2.3 which considers two Interface Automata. Since the purpose of our approach is to compute composition of a set of components, we would like to consider the compatibility for all components. Therefore, we adopted the definition of compatibility of Interface Automata and define our own definition of compatibility shown in Def. 14. Generally speaking, our compatibility requires that for all output alphabets, there are corresponding input alphabets so that every output transition can synchronize with a corresponding input transition. More specifically, first, in a component, the input and output alphabets are not allowed to have common alphabets. Second, every input alphabet of a component is distinguishable from other input alphabets of other components. The same condition for output alphabets is required too. Finally, union of all input alphabets of components is equal to the union of all output alphabets of components. Thus, the system of components defined in the approach should form a closed system so that all messages are potentially exchangeable through synchronization of components.

Definition 14 (Compatibility of IA4AD) *A set of interface automata for web services* $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$ *,*$i \in [1, n]$*, are composable if*

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

The definition of composition of IA4AD is given in Def. 15. In this work, we only concern about the result of composition of all participating services. Thus, the synchronous composition in this work is defined as directly composing all services. Therefore, the result IA4AD should have only internal transitions so there is no need of defining illegal states. Instead, we should check if there is any deadlock state to see whether behavioral mismatches exist or not.

Definition 15 (Synchronous composition of IA4AD) *Synchronous composition of a set of composable IA4AD* $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$*,* $i \in [1, n]$ *is an IA4AD:*

$$\Pi_i P_i = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$$

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.*
$A^I = A^O = \emptyset$*.*
$A^I = \bigcup_i A_i$*: finite set of alphabets.*
$\Delta \subseteq Q \times A \times Q$*: set of transition relations defined in fig 3.6.*
$q^f = (q_1^f,\ \ldots, q_i^f,\ \ldots, q_n^f)$*: accepting state.*

Example 9 *For the three services in the sessional FMUS service defined in Example 8, the synchronous composition of the three services is an IA4AD*

$$P_1 \| P_2 \| P_3 = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$$

$Q = \{q0_1, q1_1, q2_1, q3_1, q4_1\} \times \{q0_2, q1_2, q2_2, q3_2, q4_2, q5_2\} \times \{q0_3, q1_3, q2_3, q3_3, q4_3\}$

$q^0 = (q0_1, q0_2, q0_3)$

$A^I = A^O = \emptyset$*.*

$A^H = \{RawData, EndOfData, Start, Data, Complete, Ack\}$

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

Figure 3.6: Definition of transition relations in Def. 15

$$
\begin{aligned}
\Delta = \{ & \ (\ (q0_1, q0_2, qk_3), RawData, (q1_1, q1_2, qk_3)\ )\ \} \\
& \{\ (\ (q1_1, q0_2, qk_3), RawData, (q1_1, q1_2, qk_3)\ )\ \} \\
& \{\ (\ (q0_1, q2_2, qk_3), RawData, (q1_1, q3_2, qk_3)\ )\ \} \\
& \{\ (\ (q1_1, q2_2, qk_3), RawData, (q1_1, q3_2, qk_3)\ )\ \} \\
& \{\ (\ (q1_1, q2_2, qk_3), EndOfData, (q2_1, q4_2, qk_3)\ )\ \} \\
& \{\ (\ (q2_1, qk_2, q0_3), Start, (q3_1, qk_2, q1_3)\ )\ \} \\
& \{\ (\ (q3_1, qk_2, q3_3), Ack, (q4_1, qk_2, q4_3)\ )\ \} \\
& \{\ (\ (qk_1, q1_2, q1_3), Data, (qk_1, q2_2, q2_3)\ )\ \} \\
& \{\ (\ (qk_1, q1_2, q2_3), Data, (qk_1, q2_2, q2_3)\ )\ \} \\
& \{\ (\ (qk_1, q3_2, q1_3), Data, (qk_1, q2_2, q2_3)\ )\ \} \\
& \{\ (\ (qk_1, q3_2, q2_3), Data, (qk_1, q2_2, q2_3)\ )\ \} \\
& \{\ (\ (qk_1, q4_2, q2_3), Complete, (qk_1, q5_2, q3_3)\ )\ \}
\end{aligned}
$$

$$q^f = (q4_1, q5_2, q4_3)$$

To demonstrate transition relations concisely and keep them easily to be understood, we use the arbitrary index k to group transition relations with all indexes of states of corresponding component in the composite state. For example, in transition relation $(\ (q0_1, q0_2, qk_3), RawData, (q1_1, q1_2, qk_3)\ )$*,* qk_3 *in the composite state* $(q0_1, q0_2, qk_3)$ *refer to a group of states:*

$$(q0_1, q0_2, q0_3),\ (q0_1, q0_2, q1_3),\ (q0_1, q0_2, q2_3),\ (q0_1, q0_2, q3_3),\ (q0_1, q0_2, q4_3)$$

Therefore, transition relation $(\ (q0_1, q0_2, qk_3), RawData, (q1_1, q1_2, qk_3)\ )$ *expresses a group of transition relations:*

$$
\begin{aligned}
& (\ (q0_1, q0_2, q0_3), RawData, (q1_1, q1_2, q0_3)\ ), \\
& (\ (q0_1, q0_2, q1_3), RawData, (q1_1, q1_2, q1_3)\ ), \\
& (\ (q0_1, q0_2, q2_3), RawData, (q1_1, q1_2, q2_3)\ ), \\
& (\ (q0_1, q0_2, q3_3), RawData, (q1_1, q1_2, q3_3)\ ), \\
& (\ (q0_1, q0_2, q4_3), RawData, (q1_1, q1_2, q4_3)\ )
\end{aligned}
$$

One may notice that in a grouped transition relation, the arbitrary index k is expressing the state of the component which does not participate in synchronization.

3.3 Model of Adaptors

In this section, the formal definition of the model for adaptors is introduced. Basically, the model is modified from *pushdown systems* with considerations of adaptor generation with non-regular behavior. We follow the discussions in Section 3.1 and the definitions of the model of components in Section 3.2 for giving definitions of the model of adaptors in the approach. Remember now the objective is to generate an adaptor behaving like the *expected sessional adaptor* mentioned in Section 3.2 for the *sessional FMUS service* shown in Fig. 3.5.

Again, by examining the behavior of expected sessional adaptor $(?R\ !R\ ?D)^n\ ?E\ !E\ ?S\ !S\ !D^n\ ?C\ !C\ ?A\ !A),\ n > 1$ mentioned in Section 3.2, and take consideration about general cases, we should be able to find out characteristics which are essential to non-regular behavior of the expected sessional adaptor. Discussions about the observations are given as follows:

- It is easy to recognize that for every message, it comes first the reception of the message and later the delivery of the same message. For example, messages `EndOfData`, `Start`, `Complete`, and `Ack` appear in the behavior of the expected adaptor first being received and then being sent right after reception. In FMUS service, the four messages only show once in interactions so that they are only received and delivered by the adaptor once. Now we may check the rest two messages `RawData` and `Data` which appear in the behavior of the expected adaptor n times. Still, it is easy to figure out `RawData` and `Data` also appear first being received and later being sent by the expected adaptor. Furthermore, the number of appearance of `RawData` and `Data` as being received and being sent are the same. Thus, we can conclude our observation about the receiving and delivery in the behavior of the expected adaptor: (1) a messages is first being received and later being delivered by an adaptor; (2) the numbers of reception and delivery of a message are the same in the behavior of an adaptor.

- It is easy to confirm that in the behavior of the expected adaptor, all messages appeared are messages from the three services in FMUS service. This means that the expected adaptor should not create new messages. Furthermore, by examining one "round" of execution in the behavior of the expected adaptor, all messages appeared complete being received and delivered when the end of the round is reached. The number of completing being received and delivered is the same with the number of appearance. Thus, we can conclude that (1) an adaptor should not create new messages; (2) behavior of an adaptor must send all received messages when finishing a round of execution.

Thus, according to above observations, we give the following requirements for the model of adaptors:

(1) An adaptor does not generate any message by itself.

(2) An adaptor only receives messages sent from services.

(3) An adaptor only sends previously received messages.

(4) An adaptor is expected to send all received messages eventually.

From above requirements, pushdown automata model should be a good choice since the stack can satisfy these requirements. More specifically, we give some discussions about reasons of choosing pushdown automata model and finally leads to definition of the model of adaptors in the approach.

- The requirements (3) and (4) are asking for models having memory capacity. Considering behavioral mismatches with reordering of messages, a stack that works in the fashion of First-In-Last-Out (FILO) is better than a queue that works in the fashion of First-In-First-Out (FIFO). It is easy to understand that queues can not do reordering of messages. One may argue why not using more complex models such as extended automata with multiple queues, or even Tuning machines. However, more complexities lead to more difficulties in reasoning, analysis, and computation. Thus, we prefer starting from simplest types of memories, i.e., a stack.

- The requirements (1), (2), and (4) show what kind of adaptor is suitable for adaptation in service composition. This is effected by how we define service adaptation in this work. As mentioned in Section 1.3, adaptor generation without designing adaptation contracts in advance is one of the objective of this work. Therefore, we have to distinguish between parts that are able to be automated and parts that are not in adaptation contracts. Since adaptation contracts consist two parts: synchronous vectors and vector LTS, we may discuss the two cases concerning the vectors and vector LTS respectively. Vectors are basically a group of messages that the developer want participating services to interact in one or more message exchanges. In the situation that no signature mismatch exists, vectors are just pairs of messages consist of two messages of the same name except one with prefix "!" and the other with prefix "?" therefore the design of vector LTS can be simple. Developers may just follow the structure of behavior interfaces and decide how to execute all synchronizations of transitions and make sure that no deadlocks encountered. Thus, adaptor generation can be automated while no signature mismatch exist in the system of services. This means fully automated adaptor generation is possible for pure behavioral mismatching services. On the other hand, when there are signature mismatches, criteria of deciding what messages are to be mapped together in a synchronization have to take references of specifications other than just behavior interfaces. These specifications, however, include semantic or ontological descriptions in general so that manual decision is still necessary when only behavioral interfaces are available. We may conclude that in order to perform fully automated adaptor generation which is one of the objectives of this work, we assume a system of services providing behavioral interfaces with no signature mismatch. Thus, requirements (1) and (2) make sense since all messages are received and delivered by an adaptor should not be modified. Also, requirement (4) gives a condition that all received messages will and must be delivered by adaptors. This can be taken as all expected functionalities of services are fulfilled through adaptation since all messages sent are consumed with helps of an adaptor.

- All the requirements given above do not ask an adaptor to have abilities other then receiving messages and send received messages. Thus, it is easily to imagine an adaptor being represented by a pushdown automata whose set of alphabets and set of symbols are the same. We do not need to use general pushdown automata model but use *pushdown systems model* since pushdown system do not have labels but only stack symbols which simplified the model of adaptors. Furthermore, since only transitions that push and pop messages into the stack matter in an adaptor, it is easy to distinguish from push and pop transitions by only observing the changes in stack head. This means pushdown systems model is sufficient for representing adaptors in this work.

- The behavior of an adapted system of services and their adaptor with non-regular behavior is also non-regular. Therefore, ordinary verification techniques are useless, or model transformation as well as abstraction is required. By choosing pushdown systems model as the model of adaptors, verification purpose can be easily satisfied thanks for recently developed techniques of pushdown model checking mentioned in Section 2.5. Choosing pushdown systems model gives the approach advantages on both expressing non-regular behavior and model checking.

According to above discussions, we choose pushdown systems model as the model of adaptors in the approach. To cooperate with components expressed by IA4ADs, a modified pushdown system is used in the approach. We call the model of adaptors in the approach *Interface Pushdown System* (IPS) which is defined in Def. 16. Generally, an IPS is a pushdown system enhanced with notations of input and output alphabets from Interface Automata. Note that in a transition rule $(p, \gamma) \hookrightarrow (p', w)$, w does not represent the contents of the stack but only the word that replaces the head symbol of stack γ after transition is fired. Transitions of an IPS are restricted in three kinds: push, pop and internal to follow the notations in Interface Automata. Push transitions represent message reception and pop transitions represent message delivery. Internal transitions does not related to message exchange with services and the stack head symbols remains same after transitions are fired.

Definition 16 (Interface Pushdown System) *An interface pushdown system is defined as tuples:*

$$S = (Q, q^0, \Gamma, z, T, F)$$

where

Q*: finite set of states.*

q^0*: initial state.*

Γ*: finite set of stack symbols.*

z*: stack start symbol representing bottom of stack.* $z \in \Gamma$.

$T \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$*: set of transition relations.*

F*: finite set of final states.*

T *is restricted to the following three patterns:*

$< p, \gamma > \hookrightarrow < p', a\gamma >$*: push transition,*

$< p, a > \hookrightarrow < p', \epsilon >$: *pop transition,*
$< p, \gamma > \hookrightarrow < p', \gamma >$: *internal transition,*
where $a, \gamma \in \Gamma$, $a \neq z$.

Furthermore, considering an adaptor is supposed to interact with components represented by IA4ADs, we may add one constraint on an IPS defined in Def 16 for representing an adaptor. Since the condition of compatibility of IA4ADs defined in Def 14 demands that given components must form a closed system that all transitions labeled with output alphabets are expected to be synchronized with transitions labeled with corresponding input alphabets in different components. Recall that an adaptor does not generate messages, i.e., requirement (1), an IPS representing an adaptor should have its set of stack symbols as the union of all alphabets of given components represented by IA4ADs.

Similar to IA4AD, runs and accepting runs can be defined as shown in Def. 17 and Def. 18. Note that the accepting condition of accepting runs includes both the state is in final state and the stack is empty, i.e., stack head symbols is the stack start symbol z.

Definition 17 (Runs of IPS) *A finite trace* $\sigma = c_0 c_1 c_2 \dots c_k$ *is a run of an IPS* $S = (Q, q^0, \Gamma, z, T, F)$ *if* $c_0 = (q^0, z)$ *and* $\forall c_i = (q_i, w_i)$, $i \in [1, k-1]$, $\exists\, t \in T$, $t = (q_i, w_i(0)) \hookrightarrow (q_{i+1}, w)$, *where* $w_{i+1} = ww'$.

Definition 18 (Accepting Runs of IPS) *A run* $\sigma = c_0 c_1 c_2 \dots c_k$ *of an IPS* $S = (Q, q^0, \Gamma, z, T, F)$ *is an accepting run if* $c_k = (q, z)$, $q \in F$.

It should be noticed that Interface Pushdown Systems model defined in Def. 16 is pushdown systems model with constraints on transitions especially for communicating with services represented in IA4ADs. However, additional constraints are necessary for adaptors. Since the alphabets of an adaptor is the union of all alphabets of given services, we may define an adaptor as an IPS with an additional constraint on its alphabets. As shown in Def. 19, an adaptor is closely related to given services so that services represented in IA4ADs are required in defining an adaptor. Furthermore, since it is not the adaptor that decides the accepting state of an adapted system but the given services, all states are considered as final states in an adaptor. Therefore, an adaptor only decides acceptance by the accepting condition of empty stack.

Definition 19 (Adaptor as an IPS) *Given a set of composable IA4ADs:* $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f), i \in [1, n]$. *An adaptor for* P_i, $i \in [1, n]$ *is an IPS:*

$$D = (Q_D, q_D^0, \Gamma, z, T_D, F_D)$$

where $\Gamma_D = A_D \cup \{z\}$, $A_D = \bigcup_i A_i^I = \bigcup_i A_i^O$, *and* $F_D = Q_D$.

In Example 10, the definition of expected sessional adaptor is demonstrated. Comparing to the pushdown automata definition demonstrated in Example 7, The definition using IPS is more compact without loosing the expressing power on adaptors.

Example 10 (Expected Sessional Adaptor as an IPS) *The expected sessional adaptor with behavior*

$$(?R\ !R\ ?D)^n\ ?E\ !E\ ?S\ !S\ !D^n\ ?C\ !C\ ?A\ !A),\ n > 1$$

of the three services defined in Example 8 is defined as an IPS

$$D_{expected} = (Q_D, q_D^0, \Gamma, z, T_D, F_D)$$

where

- $Q_D = \{\ q0,\ q1,\ q2,\ q3,\ q4,\ q5,\ q6,\ q7,\ q8,\ q9,\ q10,\ q11,\ q12,\ q13,\ q14\ \}$.
- $q_D^0 = q0$.
- $\Gamma = \{\ RawData,\ Data,\ EndOfData,\ Complete,\ Start,\ Ack\ \} \cup \{\ z\ \}$.
- $z \in \Gamma$ *is initial symbol of stack.*
- $T_D = \{\ (q0,\ \epsilon) \hookrightarrow (q1,\ < RawData >),$
  $(q1,\ RawData) \hookrightarrow (q2,\ \epsilon),$
  $(q2,\ \epsilon) \hookrightarrow (q3,\ < Data >),$
  $(q3,\ \epsilon) \hookrightarrow (q4,\ < RawData >),$
  $(q5,\ RawData) \hookrightarrow (q5,\ \epsilon),$
  $(q5,\ \epsilon) \hookrightarrow (q3,\ < Data >),$
  $(q3,\ \epsilon) \hookrightarrow (q6,\ < EndOfData >),$
  $(q6,\ EndOfData) \hookrightarrow (q7,\ \epsilon),$
  $(q7,\ \epsilon) \hookrightarrow (q8,\ < Start >),$
  $(q8,\ Start) \hookrightarrow (q9,\ \epsilon),$
  $(q9,\ Data) \hookrightarrow (q10,\ \epsilon),$
  $(q10,\ Data) \hookrightarrow (q10,\ \epsilon),$
  $(q10,\ \epsilon) \hookrightarrow (q11,\ < Complete >),$
  $(q11,\ Complete) \hookrightarrow (q12,\ \epsilon),$
  $(q12,\ \epsilon) \hookrightarrow (q13,\ < Ack >),$
  $(q13,\ Ack) \hookrightarrow (q14,\ \epsilon)\ \}$
- $F_D = Q_D$.

Now we need to define the behavior of an adapted system which is the synchronous composition of services with their adaptor. The definition of adapted synchronous composition is shown in Def. 20. The result of composition is also an IPS. As shown in Fig. 3.7, the resulted transitions are synchronized in two cases: output transitions in services are synchronized with push transitions in coordinator; input transitions in services are synchronized with pop transitions in coordinator. For internal transitions in services, coordinator remains same state and same stack content. Note that in the composition of IA4ADs with IPS, the resulted IPS has transitions of pushing and popping stack symbols which looks like receiving and sending messages. However, these transitions should not be recognized as transitions for synchronization with services. These transitions should be considered as only replacing stack symbols in the stack head. In Example 11, we show the adapted composition of the three services in the sessional FMUS service represented

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

Figure 3.7: Definition of transition relations in Def. 20

in IA4ADs with the expected sessional adaptor we designed. We may compare the two IPS: the expected sessional adaptor and the resulted of adapted synchronous composition. In the IPS of expected sessional adaptor, the transitions of pushing stack symbols, i.e., receiving messages from services, does not have the stack head symbols specified in the left side of arrow. On the other hand, in the transition of pushing stack symbols in the resulted IPS, the stack head symbols to be replaced are all specified due to the computation of adapted synchronous composition.

Definition 20 (Adapted Synchronous Composition) *Given a set of composable IA4AD: $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$, $i \in [1, n]$, with an adaptor $D = (Q_D, q_D^0, \Gamma, z, T_D, F_D)$. The adapted synchronous composition is an interface pushdown system*

$$\Pi_i^D P_i = (Q, q^0, \Gamma, z, T', F)$$

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.*

$T' \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$*: set of transition relations defined in Fig. 3.7.*

$F = \{(q_1^f, \ \ldots, q_i^f, \ \ldots, q_n^f)\} \times F_D$*: finite set of final states.*

Example 11 (Synchronous composition with expected sessional adaptor) *The adapted synchronous composition of the three services of the sessional FMUS service defined in Example 8 with the expected sessional adaptor defined in Example 10 is an IPS*

$$S = (Q, q^0, \Gamma, z, T, F)$$

where

- $Q = Q_1 \times Q_2 \times Q_3 \times Q_D$.
- $q^0 = (q_1^0, q_2^0, q_3^0, q_D^0)$.
- $\Gamma = \{\ RawData,\ Data,\ EndOfData,\ Complete,\ Start,\ Ack\ \} \cup \{\ z\ \}$.
- $T = \{$
  $(((q0_1, q0_2, q0_3, q0_D), z) \hookrightarrow ((q1_1, q0_2, q0_3, q1_D), < RawData\ z >)),$
  $(((q1_1, q0_2, q0_3, q1_D), RawData) \hookrightarrow ((q1_1, q1_2, q0_3, q2_D), < \epsilon >)),$
  $(((q1_1, q1_2, q0_3, q2_D), z) \hookrightarrow ((q1_1, q2_2, q0_3, q3_D), < Data\ z >)),$
  $(((q1_1, q1_2, q0_3, q3_D), Data) \hookrightarrow ((q1_1, q2_2, q0_3, q4_D), < RawData\ Data >)),$
  $(((q1_1, q2_2, q0_3, q4_D), RawData) \hookrightarrow ((q1_1, q3_2, q0_3, q5_D), < \epsilon >)),$
  $(((q1_1, q3_2, q0_3, q5_D), Data) \hookrightarrow ((q1_1, q2_2, q0_3, q3_D), < Data\ Data >)),$
  $(((q1_1, q2_2, q0_3, q3_D), Data) \hookrightarrow ((q2_1, q2_2, q0_3, q6_D), < EndOfData\ Data >)),$
  $(((q2_1, q2_2, q0_3, q6_D), EndOfData) \hookrightarrow ((q2_1, q4_2, q0_3, q7_D), < \epsilon >)),$
  $(((q2_1, q4_2, q0_3, q7_D), Data) \hookrightarrow ((q3_1, q4_2, q0_3, q8_D), < Start\ Data >)),$
  $(((q3_1, q4_2, q0_3, q8_D), Start) \hookrightarrow ((q3_1, q4_2, q1_3, q9_D), < \epsilon >)),$
  $(((q3_1, q4_2, q1_3, q9_D), Data) \hookrightarrow ((q3_1, q4_2, q2_3, q10_D), < \epsilon >)),$
  $(((q3_1, q4_2, q2_3, q10_D), Data) \hookrightarrow ((q3_1, q4_2, q2_3, q10_D), < \epsilon >)),$
  $(((q3_1, q4_2, q2_3, q10_D), z) \hookrightarrow ((q3_1, q5_2, q2_3, q11_D), < Complete\ z >)),$
  $(((q3_1, q5_2, q2_3, q11_D), Complete) \hookrightarrow ((q3_1, q5_2, q3_3, q12_D), < \epsilon >)),$
  $(((q3_1, q5_2, q3_3, q12_D), z) \hookrightarrow ((q3_1, q5_2, q4_3, q13_D), < Ack\ z >)),$
  $(((q3_1, q5_2, q4_3, q13_D), Complete) \hookrightarrow ((q4_1, q5_2, q4_3, q14_D), < \epsilon >))$
  $\}$
- $F_D = \{\ (q_1^0, q_2^0, q_3^0)\ \} \times Q_D$.

Chapter 4

Detection of Behavioral Mismatches

This chapter gives details of detection of behavior mismatch. First, detection by finding deadlock states is introduced. Then detection by model checking for property of *Behavior Mismatch Free* is introduced. Also, detection of behavioral mismatches using SPIN model checker is demonstrated with the sessional FMUS service.

4.1 Detection by Finding Deadlock States

Basically, behavior mismatches can be detected only in composition of components. Therefore, we should first synchronously compose given components represented by IA4ADs to detect existence of behavior mismatches. As mentioned in Section 2.1 and Section 2.2, behavior mismatches are confirmed exist when deadlock states are found in the synchronous composition of given components. Therefore, detection of behavior mismatches are basically have two steps:

- Compute synchronous composition of components.
- Check existence of deadlock states in the above composition.

For given components represented by IA4ADs, the synchronous composition is defined in Def. 15 We may call the synchronous composition of components the system behavior since this composition is the behavior of the system if there is no behavior mismatch in the components. Furthermore, recall in the overview of the approach described in Section 1.4, given behavior interfaces of components are supposed to pass the compatibility check. The condition of compatibility of components defined in Def. 14 generally means all messages can be potentially synchronized when the behavior interfaces of components are given. Thus, signature mismatches are not exist in given behavior interfaces since names of messages are already confirmed matching. Mismatches in the system behavior of given components are only behavior mismatches.

To demonstrate with the sessional FMUS service, recall that we have showed the synchronous composition of IA4ADs of the three services in Example 9. The synchronous composition gives the system behavior of the FMUS services and the first step of detection of behavior mismatch is done. We may proceed to the second step to check the existence of deadlock states. Basically, deadlock states can be defined as states that are reachable from the initial state but can not reach the final state in the system behavior. Def. 21 defines deadlock states in an IA4AD. It should be noticed that states not reachable from

the initial state is ignored. It is possible that such a state exists in the system behavior where no deadlock state exists. States not reachable from the initial state are never reached in the system behavior and effect nothing in the system.

Definition 21 (Deadlock states of an IA4AD) *Deadlock states of an IA4AD* $P = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$ *are states reachable from the initial state* q^0 *but can not reach the final state* q^f*:*

$$Q^{Deadlock} \;=\; \{\; q \mid \exists q^0 \rightarrow^* q \;\} \;\setminus\; \{\; q \mid \exists q \rightarrow^* q^f \;\}$$

where $q_0 \rightarrow^* q_n$ *is a sequence of transition relations* $\delta_1\delta_2 \ldots \delta_n$ *connecting* q_0 *and* q_n*:* $\delta_1 = (q_0, a1, q_1)$, $\delta_2 = (q_1, a1, q_2)$, $\ldots$, $\delta_n = (q_{n-1}, a1, q_n)$.

According to Def. 21, we may check the synchronous composition in Example 9 and conclude that all states in the synchronous composition are deadlock states. Only state $(q1_1, q1_2, q0_3)$ is reachable from the initial state and all states can not reach the final state. Therefore, $(q1_1, q1_2, q0_3)$ is the deadlock state of the system behavior of the sessional FMUS service and we can confirm the existence of behavioral mismatches. Algorithm of searching for deadlock states in an IA4AD is quite intuitive and not necessary to be mentioned here. Instead, we would like to introduce a different computation using model checking technique which will be given in Section 4.2.

4.2 Detection by Model Checking

The definition of deadlock states in an IA4AD shown in Def. 21 may be represented by another form using traces of executions of an IA4AD. Recall that we have defined *acceptance runs of an IA4AD* in Def. 13. Comparing to definitions of deadlock states and acceptance runs, we can conclude that if there is a state which is not a deadlock, there is at least an acceptance run corresponding to the state. Furthermore, if there is no deadlock state in an IA4AD, then all runs of the IA4AD are acceptance runs. We define this condition in Def. 22 and call this condition *Behavior Mismatch Free*.

Definition 22 (Behavior Mismatch Free of an IA4AD) *Behavior Mismatch Free of an IA4AD* $P = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$ *is all runs of* P *are acceptance runs.*

The problem of determining Behavior Mismatch Free of an IA4AD is basically a reachability check problem which checks the reachability to the final state of the IA4AD. Since model checking algorithms are generally searching for reachability of specified states, it is intuitive to apply model checking techniques on detection of behavioral mismatches.

In order to apply LTL model checking introduced in Section 2.4, we need to prepare the following:

1. A transition system M.
2. Atomic propositions AP.
3. labeling function L.

By Def. 15, we can compute the synchronous composition, i.e., the system behavior, of given components represented by IA4ADs so preparation of M is done. For the atomic propositions AP, we only need a proposition which represents the acceptance condition and denoted it as p_{accept}. Finally we need to define a labeling function which assigns propositions in AP to states of M,i.e., the IA4AD. The answer is intuitive and simple that p_{accept} is assigned only to the final sate q^f. Thus, we define a LTL property which represents the property of *Behavior Mismatch Free* in Def. 23.

Definition 23 (Property of Behavior Mismatch Free of an IA4AD) *Given an IA4AD: $P = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$, the property of Behavior Mismatch Free is written as a LTL formula*

$$\Diamond p_{accept}$$

where p_{accept} is an atomic proposition. The labeling function for state $q \in Q$ is defined as

$$L(q) : \{p_{accept} \mid q = q^f\}$$

If a system behavior, i.e., synchronous composition of given IA4ADs which is also an IA4AD, passes the property of Behavior Mismatch Free in model checking, the system behavior is then declared has no behavioral mismatch since we can not find a trace that is not an acceptance run in the system. Otherwise, if the model checking for the property of Behavior Mismatch Free fails by returning a counterexample, there exist behavior mismatches in the system behavior and therefore an adaptor is needed in order to compose given components. It should be noticed that a counterexample is a trace that violates the specified property. In the case of the property of Behavior Mismatch Free, a counterexample shows a trace that starts from the initial state but ends at a state which is not the final state. Thus, the states where the trace of the counterexample stops is simply a deadlock state.

Improvements of model checking techniques make it easy to apply model checking in the approach. We can apply model checking using model checkers such as SPIN [7]. SPIN is one of the most popular model checkers. The input model Promela is similar to C programs which is easy to learn and apply. Thus, instead of directly applying algorithms of LTL model checking, we use SPIN model checker. The only problem is how to encode the Promela model for SPIN. Practically, behavior interfaces of components represented by IA4ADs can be implemented in Promela as a system of processes communicating through one synchronous message queue. Then SPIN can do both synchronous composition and model checking for us. If the system behavior passes the check, there will be no counterexample returned. This means the given components works well by themselves and no adaptation is needed. Otherwise, there will be a returned counterexample showing a trace leading to a deadlock state. In this case, adaptation is needed and we proceed to the next step of adaptor generation.

It should be noticed that the detection of behavioral mismatches on software components represented in LTSs is already proposed in the conventional framework. The way of detection of behavior mismatches in the conventional framework is similar to the computation introduced in Section 4.1. However, in this work, the approach intentionally uses model checking for the purpose of integrating verification processes in adaptation. This means we may specify other properties and apply model checking for these properties and

the property of Behavior Mismatch Free. Therefore, verification purpose could also be fulfilled in the process of detecting behavior mismatches.

Here the session FMUS service shown in Fig. 3.5 is used to demonstrate how to use SPIN to detect behavior mismatches. Recall that the three services of the sessional FMUS service are defined as IA4ADs in Example 8, and their synchronous composition, i.e., the system behavior, is shown in Example 9. To use SPIN, we do not encode the system behavior into Promela model directly but instead encoding a system of three components synchronously communicating through a synchronous channel. The Promela model includes four parts parts. The first part defines some basic settings which can be further divided into three parts. The first setting defines the acceptance propositions of the three services. These propositions are defined as state indexes of active states where the state indexes are the numbers assigned for the final states of corresponding services. These propositions will be used also in building LTL formula of the property of behavior mismatch free. The second setting defines the set of messages in `mtype` so that we may treat each name as an entity in communication. The third setting defines a synchronous queue. When we set the length of a queue to zero in Promela, the queue becomes synchronous since it can hold only one element so that no new element can be put into the queue until the held element is consumed. The second and third settings define the synchronous communication environment in the Promela model.

```
#define accept_OnlineStockBroker (OnlineStockBroker:active_state==4)
#define accept_ResearchDepartment (ResearchDepartment:active_state==5)
#define accept_Investor (Investor:active_state==4)

/* define names of messages */
mtype = {RawData, EndOfData, Start, Data, Complete, Ack};

/* one synchronous channel for all services */
chan sync_chan = [0] of { mtype };
```

The second part of the Promela code is the definition of behavior interfaces of the three services. The behavior of a services is defined as a process in Promela. Since the behavior interface of a service is defined in IA4AD, it is intuitive to encode the behavior of the three services in the following fashion. First states are represented as numbers assigned to an variable `active_state` which defines the current active state of a service. Thus, the transitions are executed by putting or getting messages from the synchronous queue then assigning a state number to `active_state`. It should be noticed that the putting and getting messages from a queue in Promela is written using special symbols "!" and "?" which shares the same meaning in representing output and input transitions of services.

```
/* interface protocol of OnlineStockBroker */
proctype OnlineStockBroker(chan q) {
short active_state;
active_state = 0; /* initial state */
```

```
do
:: (active_state == 0) ->
    if
    :: q!RawData -> active_state = 1
    fi;
:: (active_state == 1) ->
    if
    :: q!RawData -> active_state = 1
    :: q!EndOfData -> active_state = 2
    fi;
:: (active_state == 2) ->
    if
    :: q!Start -> active_state = 3
    fi;
:: (active_state == 3) ->
    if
    :: q?Ack -> active_state = 4
    fi;
od
}
```

```
/* interface protocol of ResearchDepartment */
proctype ResearchDepartment(chan q) {
short active_state;
active_state = 0; /* initial state */

do
:: (active_state == 0) ->
    if
    :: q?RawData -> active_state = 1
    fi;
:: (active_state == 1) ->
    if
    :: q!Data -> active_state = 2
    fi;
:: (active_state == 2) ->
    if
    :: q?RawData -> active_state = 3
    :: q?EndOfData -> active_state = 4
    fi;
:: (active_state == 3) ->
    if
    :: q!Data -> active_state = 2
    fi;
:: (active_state == 4) ->
```

```
    if
    :: q!Complete -> active_state = 5
    fi;
od
}
```

```
/* interface protocol of Investor */
proctype Investor(chan q) {
short active_state;
active_state = 0; /* initial state */

do
:: (active_state == 0) ->
    if
    :: q?Start -> active_state = 1
    fi;
:: (active_state == 1) ->
    if
    :: q?Data -> active_state = 2
    fi;
:: (active_state == 2) ->
    if
    :: q?Data -> active_state = 2
    :: q?Complete -> active_state = 3
    fi;
:: (active_state == 3) ->
    if
    :: q!Ack -> active_state = 4
    fi;
od
}
```

The third part sets the initial state of the system described in Promela. The initial setting is a process which assigns all processes, i.e., behavior of services, the queue for communication then call the start of all processes. Note that the queue assigned for the three processes is the synchronous queue define in the first part of Promela.

```
/* init process */
init {
  run OnlineStockBroker(sync_chan);
  run ResearchDepartment(sync_chan);
  run Investor(sync_chan);
}
```

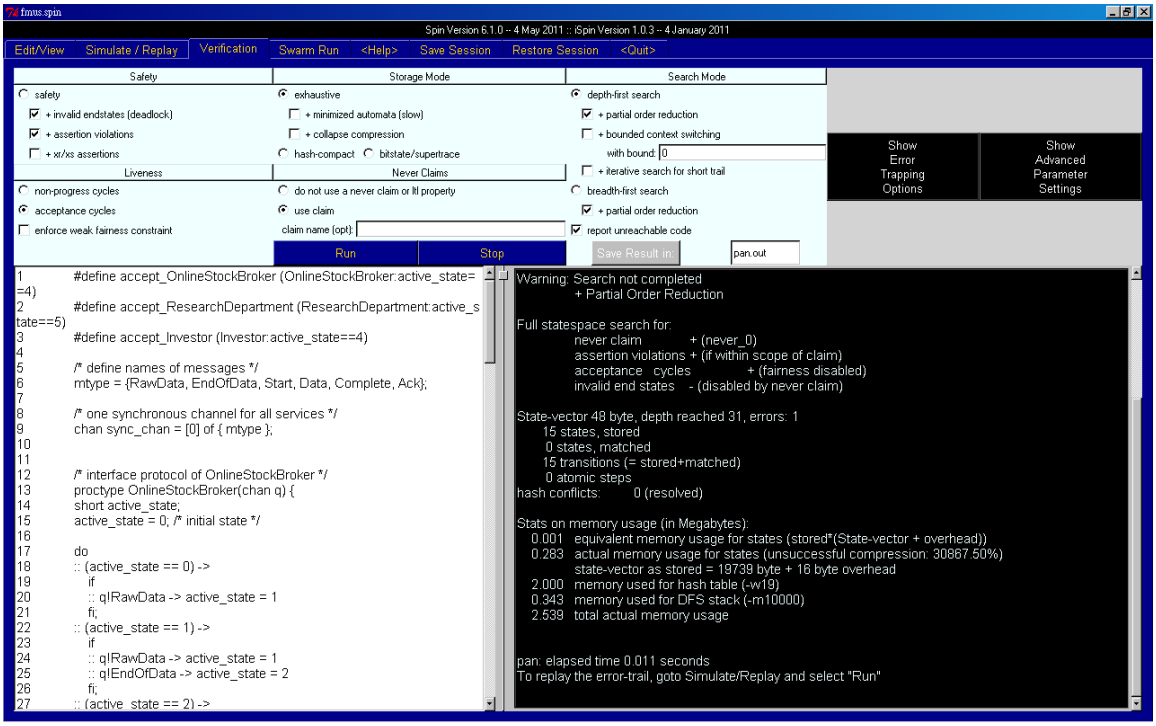


Figure 4.1: FMUS service: Detection of Behavior Mismatches by SPIN - Verification Screen

The detection of behavioral mismatches using model checking by SPIN also need a LTL formula representing the property of behavior mismatch free. The LTL formula is defined below and is constructed by connecting the three acceptance propositions of the three services. Then the negation of the formula is transformed to a Büchi automaton call never claim by SPIN. Then SPIN performs LTL model checking to check the system for the property of behavior mismatch free.

```
<> ( accept_OnlineStockBroker &&
          accept_ResearchDepartment &&
                accept_Investor       )
```

The verification of the Promela model of the FMUS service is shown in Fig. 4.1. SPIN provides graphical user interface called *iSPIN* for developers to conduct verifications easier. The result shows some technical information including that a violation to the property of Behavior Mismatch Free is found and output as a trail for further use. iSPIN also provides a simulation mode for demonstrating generated trails. As shown in Fig. 4.2, iSPIN can read the trail information and generate a graph for us. For mode details, the graph of the generated trail is shown in Fig. 4.3. It is easy to confirm that only the first synchronization between `Online Stock Broker` and `Rreserach Department` is executed and there is no further execution. Thus, we may conclude that the FMUS service encoded in Promela model fails to satisfy the property of Behavior Mismatch Free, which confirms the existence of behavior mismatches.

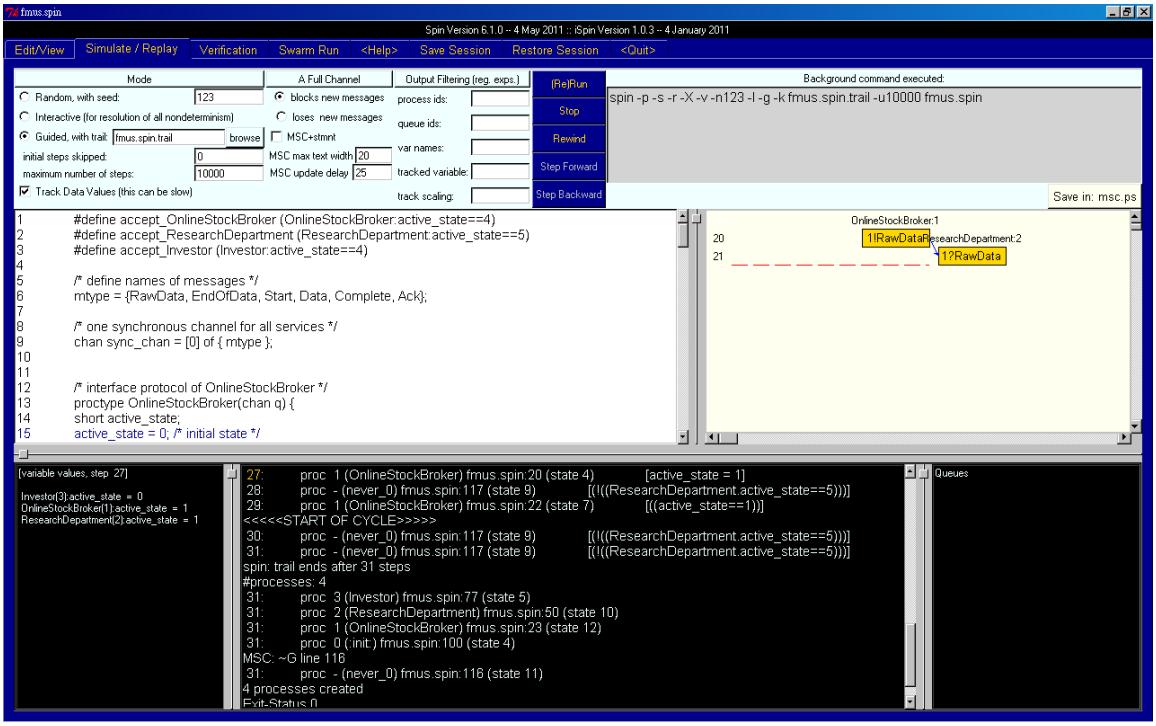


Figure 4.2: FMUS service: Detection of Behavior Mismatches by SPIN - Trail Simulation Screen

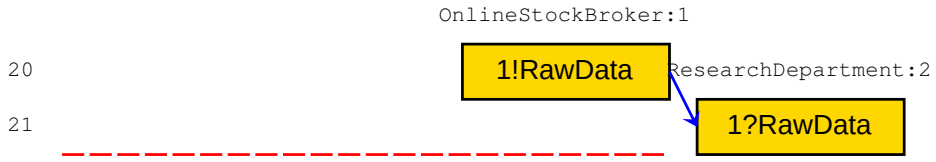


Figure 4.3: FMUS service: Detection of Behavior Mismatches by SPIN - The trail

Chapter 5

Coordinator Guided Adaptor Generation

5.1 Overview of Adaptor Generation

Recall that in Section 3.1, components are required to be expressed in the fashion of *one session process*. We may start to skip the word *sessional* in the rest of this thesis for convenience. Thus, the FMUS service means the system shown in Fig. 3.5 from now on. Also, when demonstrating adaptor generation using the FMUS service, it may become too complicate to show the demonstration in full detail. Therefore, a much simpler example consists of two services Sender and *Receiver* shown in Fig. 5.1 is introduced. We may call this simple example *SR service* in short for convenience. In the SR service, *Sender* sends three messages `a`, `b` and `c` to *Receiver*. It is easy to recognize that the behavior interfaces of *Sender* and *Receiver* causes a behavior mismatch. The ordering of messages `a` and `b` being received by *Receiver* is reversed to the ordering being sent by *Sender*. Also the self transitions of sending and receiving `c` in the two services causes non-regular behavior in adaptors. The expected adaptor is designed to have behave like the languages

$$?a\ ?c^n\ ?b\ !b\ !c^n\ ?a$$

.

The adaptor generation in the approach is called *Coordinator Guided Adaptor Generation*. The overview of adaptor generation is shown in Fig. 5.2. The core idea is as follows:

- An over-behavioral adaptor called *Coordinator* is first generated from given components represented by IA4ADs. Coordinator is then composed with given components which give us an Interface Pushdown System, i.e., the system behavior with Coordinator. The system behavior is also over-behavioral because of Coordinator.
- Pushdown model checking is applied to the above system behavior, an IPS. We may assume a property ϕ represents the requirements of an adaptor for given components. In the approach, the pushdown model checking is performed for the *negation* of ϕ. This gives us a counterexample which is a trace satisfies ϕ and therefore is a candidate of adaptors for the system.

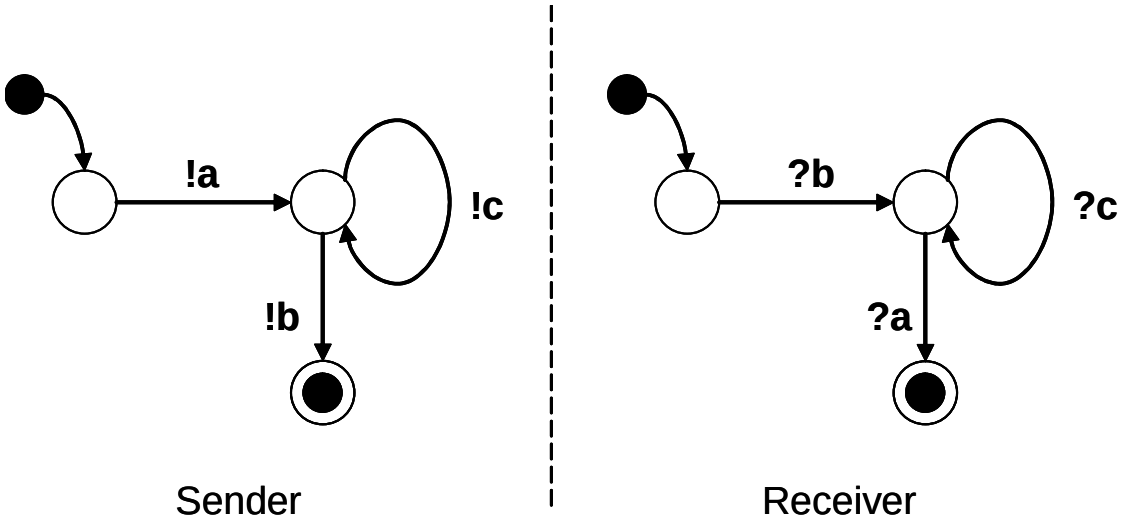


Figure 5.1: SR service

- The property ϕ is the most essential part in adaptor generation of the approach. We need property of *Behavior Mismatch Free* and property for *Unbounded Messages* to guarantee the correctness of an adaptor. The former is for solving behavior mismatches and the later is for non-regular behavior of adaptors.
- If there exist adaptors for given components, pushdown model checking for negation of ϕ should return a counterexample. The counterexample is a trace that satisfies the property ϕ and is used to build an adaptor as and IPS.

Details of each step of adaptor generation in the approach will be described in the following sections along with demonstrations with the SR service and FMUS service.

5.2 Coordinator

First, Coordinator is a special adaptor which is capable of (1) receiving any message sent from a component in the system; (2) sending any message happen to be the head symbol of the stack to a component which is ready to receive it. Thus, we say that the behavior of Coordinator is *over-behavioral*. This means Coordinator provides the maximum of possible interactions of given components under adaptation. To build Coordinator for given components, it is easy to imagine an IPS with only one state that has all combinations of pushing and popping transitions. The definition of coordinator is shown in Def. 24. Note that it does not matter how the coordinator behave so Coordinator only has one state. It only matters that Coordinator has all combinations of pushing and popping transitions for all messages in the system of given components.

Definition 24 (Coordinator) *Given a set of composable IA4ADs* $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$, $i \in [1, n]$. *A coordinator is an IPS* $C = (Q_C, q_C^0, \Gamma, z, T_C, F_C)$, *where*

$Q_C = \{q_C^0\}$ *is the finite set of states has only the initial state* q_C^0;

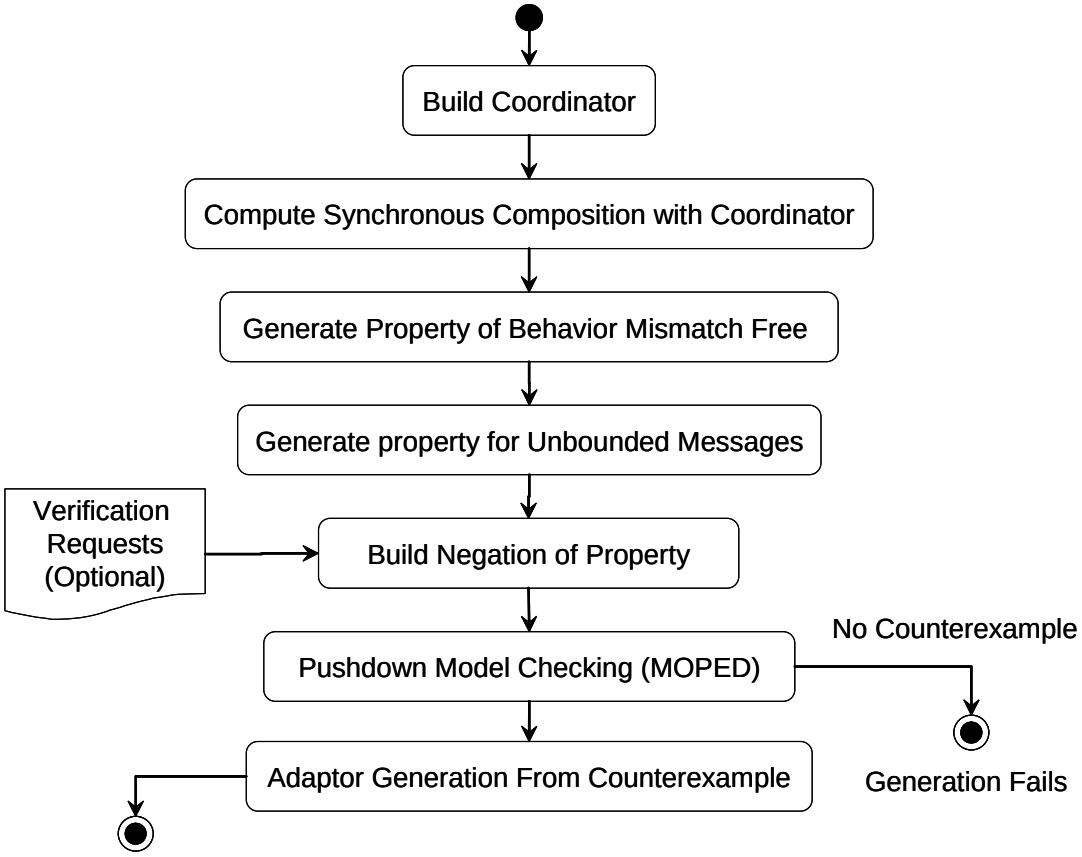


Figure 5.2: Overview of adaptor generation

$A_C = \bigcup_i A_i^I = \bigcup_i A_i^O$ *is the finite set of alphabets;*

$\Gamma = A_C \cup \{z\} \cup \{\epsilon\}$ *is the finite set of stack symbols;*

z *is the stack start symbol representing bottom of stack;*

$F_C = Q_C$ *is the finite set of final states.*

$T_C = (Q_C \times A_C \times \Gamma) \times (Q_C \times \Gamma^*)$ *is the set of transition relations defined as follows:*

$(q_C^0, \gamma) \hookrightarrow (q_C^0, a\gamma)$ *is a push transition,*
$(q_C^0, a) \hookrightarrow (q_C^0, \epsilon)$ *is a pop transition,*
where $a \in A_C$, $\gamma \in \Gamma$ *is the head symbol of stack.*

The synchronous composition of components and their Coordinator is also over-behavioral which has both *good behavior*, i.e., behavior with no behavioral mismatch, and *bad behavior*, i.e., behavior with behavioral mismatches. By the definition of adapted synchronous composition in Def. 20 and the definition of Coordinator in Def. 24, we can compute the synchronous composition of components and their Coordinator, the *over-behavioral* IPS. The *over-behavioral* IPS should contain two kinds of behavior: behavior consists of traces can reach the final state, i.e., the good behavior, and behavior consists of traces can not reach the final state, i.e., the bad behavior. The criteria of distinguishing good and bad behavior in the over-behavioral IPS is Behavior Mismatch Free. It should be noticed that any trace in the good behavior could be the behavior an adaptor for given components. Thus, the objective of adaptor generation in the approach is to pick up traces in good behavior and build an adaptor for the system of given components.

Example 12 demonstrates Coordinator for the FMUS services. In the Coordinator, there is only one state and transitions are all self transitions. The self transitions can be classified into two categories: pushing and popping. The pushing transitions can push any stack symbol in Γ. The stack head symbols to be replaced in pushing transitions are all ϵ which means the pushing transition adds a stack symbol γ to the stack no matter what stack symbol is in the head of the stack. Therefore, there are six pushing transitions in Coordinator for FMUS service since there are six different messages in the system. On the other hand, there are six popping transitions in which the stack head symbols to be replaced are the six messages respectively. The replacing stack symbol in a popping transition is ϵ which means the deletion of the stack head symbol in the left side of the transition.

Example 12 (Coordinator for FMUS service) *Coordinator for the three services defined in Example 8 for the FMUS services shown in Fig. 3.5 is defined as an IPS*

$$C = (Q_C, q_C^0, \Gamma, z, T_C, F_C)$$

where

$Q_C = \{\ q0_C\ \}$.

$q_C^0 = q0_C$.

$\Gamma = \{\ RawData,\ Data,\ EndOfData,\ Complete,\ Start,\ Ack\ \} \cup \{\ z\ \}$.

$z \in \Gamma$ *is initial symbol of stack.*

$$
\begin{aligned}
T_C = \{\ & (q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < RawData >),\\
& (q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < Data >),\\
& (q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < EndOfData >),\\
& (q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < Complete >),\\
& (q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < Start >),\\
& (q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < Ack >),\\
& (q0_C,\ RawData) \hookrightarrow (q0_C,\ \epsilon),\\
& (q0_C,\ Data) \hookrightarrow (q0_C,\ \epsilon),\\
& (q0_C,\ EndOfData) \hookrightarrow (q0_C,\ \epsilon),\\
& (q0_C,\ Complete) \hookrightarrow (q0_C,\ \epsilon),\\
& (q0_C,\ Start) \hookrightarrow (q0_C,\ \epsilon),\\
& (q0_C,\ Ack) \hookrightarrow (q0_C,\ \epsilon)\ \}
\end{aligned}
$$

$F_C = Q_C$.

The adapted synchronous composition of the three services in the FMUS service with coordinator will have incredibly more transitions than the composition with expected adaptor shown in Example 11. Since coordinator provides all combinations for synchronization with given components, the number of transition relations should reach more than the number of one hundred and fifty. It is easy to calculate and get this number since there are five, six, and five states in each of the three service that have transitions being able to synchronize with coordinator, i.e., $5 \times 6 \times 5 = 150$. This number should further times for number of services and adds the counts of multiple transitions from a state. Therefore, the list of full transition relations in the composition with coordinator for FMUS service is skipped. Instead we demonstrate with the SR service shown in Fig. 5.1. Example 13 demonstrates the over-behavioral system behavior from synchronous composition of services *Sender* and *Receiver* with their Coordinator. One may count the number of transition relations and get a number eighteen.

Example 13 (Synchronous composition with coordinator for SR service) *The two services shown in Fig. 5.1 is represented in IA4AD as follows:*

Sender*:*
$P_1 = (\ Q_1,\ q_1^0,\ A_1^I,\ A_1^O,\ A_1^H,\ \Delta_1,\ q_1^f)$ *where*

$Q_1 = \{\ q0_1,\ q1_1,\ q2_1\ \}$.
$q_1^0 = q0_1$.
$A_1^I = \emptyset$.
$A_1^O = \{\ a,\ b,\ c\ \}$.
$A_1^H = \emptyset$.
$q_1^f = q2_1$.
$\Delta_1 = \{\ (q0_1,\ !a,\ q1_1),\ (q1_1,\ !c,\ q1_1),\ (q1_1,\ !b,\ q2_1)\ \}$.

Receiver*:*
$P_2 = (\ Q_2,\ q_2^0,\ A_2^I,\ A_2^O,\ A_2^H,\ \Delta_2,\ q_2^f)$ *where*

$Q_2 = \{\ q0_2,\ q1_2,\ q2_2\ \}.$
$q_2^0 = q0_2.$
$A_2^I = \{\ a,\ b,\ c\ \}.$
$A_2^O = \emptyset.$
$A_2^H = \emptyset.$
$q_2^f = q2_2.$
$\Delta_1 = \{\ (q0_2,\ ?b,\ q1_2),\ (q1_2,\ ?c,\ q1_2),\ (q1_2,\ ?a,\ q2_2)\ \}.$

Then Coordinator for the SR service $C = (Q_C, q_C^0, \Gamma, z, T_C, F_C)$ *is designed as follows:*

$Q_C = \{\ q0_C\ \}.$
$q_C^0 = q0_c.$
$\Gamma = \{\ a,\ b,\ c\ \} \cup \{\ z\ \}.$
$z \in \Gamma$ *is initial symbol of stack.*
$T_C = \{\ (q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < a >),$
$(q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < b >),$
$(q0_C,\ \epsilon) \hookrightarrow (q0_C,\ < c >),$
$(q0_C,\ a) \hookrightarrow (q0_C,\ \epsilon),$
$(q0_C,\ b) \hookrightarrow (q0_C,\ \epsilon),$
$(q0_C,\ c) \hookrightarrow (q0_C,\ \epsilon)\ \}$
$F_C = Q_C.$

Thus, the adapted synchronous composition of the two services with their coordinator is computed and shown as an IPS $S = (Q, q^0, \Gamma, z, T, F)$ *where*

$Q = Q_1 \times Q_2 \times Q_C.$
$q^0 = (q0_1, q0_2, q0_C).$
$\Gamma = \{\ a,\ b,\ c\ \} \cup \{\ z\ \}.$
$T = \{$
$(((q0_1, q0_2, q0_C), \epsilon) \hookrightarrow ((q1_1, q0_2, q0_C), < a >)),$
$(((q0_1, q0_2, q0_C), b) \hookrightarrow ((q0_1, q1_2, q0_C), \epsilon)),$
$(((q0_1, q1_2, q0_C), \epsilon) \hookrightarrow ((q1_1, q1_2, q0_C), < a >)),$
$(((q0_1, q1_2, q0_C), c) \hookrightarrow ((q0_1, q1_2, q0_C), \epsilon)),$
$(((q0_1, q1_2, q0_C), a) \hookrightarrow ((q0_1, q2_2, q0_C), \epsilon)),$
$(((q0_1, q2_2, q0_C), \epsilon) \hookrightarrow ((q1_1, q2_2, q0_C), < a >)),$
$(((q1_1, q0_2, q0_C), \epsilon) \hookrightarrow ((q1_1, q0_2, q0_C), < c >)),$
$(((q1_1, q0_2, q0_C), \epsilon) \hookrightarrow ((q2_1, q0_2, q0_C), < b >)),$
$(((q1_1, q0_2, q0_C), b) \hookrightarrow ((q1_1, q1_2, q0_C), \epsilon)),$
$(((q1_1, q1_2, q0_C), \epsilon) \hookrightarrow ((q1_1, q1_2, q0_C), < c >)),$
$(((q1_1, q1_2, q0_C), \epsilon) \hookrightarrow ((q2_1, q1_2, q0_C), < b >)),$
$(((q1_1, q1_2, q0_C), c) \hookrightarrow ((q1_1, q1_2, q0_C), \epsilon)),$
$(((q1_1, q1_2, q0_C), a) \hookrightarrow ((q1_1, q2_2, q0_C), \epsilon)),$
$(((q1_1, q2_2, q0_C), \epsilon) \hookrightarrow ((q1_1, q2_2, q0_C), < c >)),$
$(((q1_1, q2_2, q0_C), \epsilon) \hookrightarrow ((q2_1, q2_2, q0_C), < b >)),$

$$
\begin{aligned}
&(((q2_1, q0_2, q0_C), b) \hookrightarrow ((q2_1, q1_2, q0_C), \epsilon)),\\
&(((q2_1, q1_2, q0_C), c) \hookrightarrow ((q2_1, q1_2, q0_C), \epsilon)),\\
&(((q2_1, q1_2, q0_C), a) \hookrightarrow ((q2_1, q2_2, q0_C), \epsilon)),\\
&\}\\
F = \{\ &(q2_1, q2_2, q0_C)\ \}.
\end{aligned}
$$

5.3 Behavior Mismatch Free

For detection of behavior mismatches in Section 4.2, the property of Behavior Mismatch Free for the system of components is already given in Def. 23. This property is also necessary for adaptor generation in the approach. However, the property is not suitable for the synchronous composition of components with Coordinator, i.e., an adaptor, because of the stack in the composition. Therefore, we need to redefine the property of Behavior Mismatch Free for the IPS computed from adapted synchronous composition with Coordinator. There is no need to redefine the proposition p_{accept} itself. Instead the labeling function needs to be redefined for taking the stack into consideration. Recall the acceptance condition of an IPS defined in Def. 18, the condition of empty stack is necessary. Therefore we redefine the labeling function by adding the empty stack condition. The redefined property of Behavior Mismatch Free is shown in Def. 25. Note that $\gamma = z$ is added to express the condition of empty stack since z is the start stack symbol. Also note that the labeling function is defined for a pair consists of the state and the head stack symbol.

Definition 25 (Property of Behavior Mismatch Free for IPS) *Given an IPS* $S = (Q,\ q^0, \Gamma, z,\ T, F)$*, the property of Behavior Mismatch Free is written as a LTL formula*

$$\Diamond p_{accept}$$

p_{accept} *is an atomic proposition and a labeling function for state* s *and stack head* γ *is defined as*

$$L((s, \gamma)) : \{p_{accept} \mid s \in F\ \wedge\ \gamma = z\}$$

where z *is the start symbol of stack.*

Since all given components have only one final state which goes to no other state and there is only one state in Coordinator, the synchronous composition with coordinator has only one composite final state. Thus, condition of p_{accept} can be expressed by the conjunction of propositions that all IA4ADs is at their final states and the empty stack. The definition is given in Def. 26.

Definition 26 (Property of Adapted Behavior Mismatch Free) *Given an IPS* $S = (Q,\ q^0, \Gamma, z,\ T, F)$ *which is the adapted synchronous composition of a set of composable IA4ADs* $P_i = (Q_i, q_i^0,\ A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f), i \in [1, n]$ *and their Coordinator* $C = (Q_C, q_C^0, \Gamma, z, T_C, F_C)$*, the property of Behavior Mismatch Free is written as a LTL formula*

$$\Diamond p_{accept}$$

p_{accept} is an atomic proposition and a labeling function for composite state $(q_1, q_2, \ldots, q_n, q_C^0)$ and stack head γ is defined as

$$L(((q_1, q_2, \ldots, q_n, q_C^0), \gamma)) : \{p_{accept} \mid \bigwedge_{i=1\ldots n} q_i = q^f \ \wedge \ \gamma = z\}$$

where z is the start symbol of stack.

Here we demonstrate the property of Behavior Mismatch Free for the FMUS service and SR service in Example 14.

Example 14 (Property of Adapted Behavior Mismatch Free) *For the FMUS service shown in Fig. 3.5, we may define the property of Behavior Mismatch Free using information in Example 8 and Example 12. The LTL formula is $\Diamond p_{accept}$ where the labeling function is*

$$L((q_1, q_2, q_3, q_C^0), \gamma) = \{\ p_{accept} \mid q_1 = q_1^f \ \wedge \ q_2 = q_2^f \ \wedge \ q_3 = q_3^f \ \wedge \ \gamma = z\ \}$$

Similarly, for the SR service shown in Fig. 5.1, we may define the property of Behavior Mismatch Free using information in Example 13. The LTL formula is still $\Diamond p_{accept}$ where the labeling function is

$$L((q_1, q_2, q_C^0), \gamma) = \{\ p_{accept} \mid q_1 = q_1^f \ \wedge \ q_2 = q_2^f \ \wedge \ \gamma = z\ \}$$

5.4 Unbounded Messages

In the non-regular behavior of the expected adaptor $(?R\ !R\ ?D)^n\ ?E\ !E\ ?S\ !S\ !D^n\ ?C\ !C\ ?A\ !A),\ n > 1$ introduced in Section 3.2, the arbitrary natural number n indicating the numbers of messages `RawData` and `Data` being sent and received are the same. The reason for the equality is not because of the expected adaptor makes the numbers the same but because of the structures of behavior interfaces of the three services which are decided by the functionalities the services are supposed to achieve. Therefore in other cases there may be one analyzed `Data` after receiving two `RawData`, which makes number related to `RawData` and `Data` $2n$ and n. However, though the equality of natural numbers is actually decided by components themselves, it is the arbitrary natural number n, instead of just $*$, that characterizes non-regular behavior in adaptors. More specifically, if there is no such arbitrary natural number in the behavior of an adaptor, LTS model is already enough since the behavior is regular in this case. Thus, it is essential to make sure the arbitrary natural number n appear whenever the system of services has an adaptor with non-regular behavior. Note that this arbitrary natural number n indicates the corresponding message sent and received arbitrary multiple times. We may call the messages being sent/received arbitrary multiple times *Unbounded Messages* and informally define the set of *Unbounded Messages* as

$$A^{UB} = \{a \mid a \in A \ \wedge \ \forall\sigma, \exists\sigma', Occ(\sigma', a) > Occ(\sigma, a)\}$$

where σ and σ' are accepting traces of the system behavior and $Occ(\sigma, a)$ represents the number of occurrence of message a in σ. The meaning of this definition says that for a

message a, given any trace that a occurs n times, there is always another trace in which a occurs more than n times.

To clarify the specification of *unbounded messages*, we may first try to define *unbounded messages* formally. Recall that runs and accepting runs of an IPS are already defined in Def. 17 and Def. 18. We may start from tracing stack symbols in accepting runs to define number of occurrences of a message. Accepting stack traces of accepting runs of an IPS can be defined as Def. 27 by extracting stack head symbols in configurations of an accepting run. By counting numbers of a specific message in a specific accepting stack trace, we can define the function of number of occurrences in Def. 28. Note that messages are abbreviated in stack symbols. Therefore, the formal definition of unbounded messages can be defined as shown Def. 29.

Definition 27 (Accepting Stack Trace of IPS) *Given a finite trace $\sigma = c_0c_1c_2 \dots c_k$ which is an accepting run of an IPS $S = (Q, q^0, \Gamma, z, T, F)$, $c_i = (q_i, w_i)$. The corresponding stack trace of σ is $w = \gamma_0\gamma_1 \dots \gamma_k$, where $\gamma_i = w_i(0)$.*

Definition 28 (Number of Occurrences) *Given an IPS $S = (Q, q^0, \Gamma, z, T, F)$, the number of occurrences of stack symbol $a \in \Gamma$ in an accepting run σ is defined as a function*

$$Occ(\sigma, a) = \mid \{\ \gamma_i \mid i \in [1, k],\ w_i(0) = a\ \} \mid$$

Definition 29 (Unbounded Messages) *Given an IPS $S = (Q, q^0, \Gamma, z, T, F)$, the set of unbounded stack symbols, i.e., Unbounded Messages is defined as*

$$\Gamma^{UB} = \{\ a \mid a \in \gamma\ \wedge\ \forall\sigma,\ \exists\sigma',\ \sigma, \sigma' \in L(S),\ Occ(\sigma', a) > Occ(\sigma, a)\ \}$$

where σ and σ' are accepting runs of S.

Following the definition of Unbounded Messages in Def. 29, locating Unbounded Messages has to examine all traces exhaustively of an IPS for the numbers of occurrences of all messages. This is unrealistic since an IPS has potentially infinite traces because of unbounded stack length. Thus, we would like to use another strategy for locating Unbounded Messages in the approach. Since an IPS, i.e., the system behavior computed from adapted synchronous composition of components with Coordinator, has only finite number of states, while traces of an IPS that has some messages occur arbitrary multiple times imply the length of some traces may be infinite. We may conclude that messages occur arbitrary multiple times are resulted by loops in the IPS. Therefore, locating *Unbounded Messages* is considered the same task of finding out loops in an IPS. Furthermore, the IPS representing the system behavior is composed from components and Coordinator. Considering composite states in the IPS, loops in traces of the IPS are also loops in each component of the system. Since Coordinator has only one state, finding loops in given components simply meets the purpose of finding loops in the IPS representing system behavior.

Now we go for techniques of finding loops in components represented by IA4ADs. We would like to introduce the idea by locating strongly connected components (SCCs) using

the Tarjan's algorithm [16]. Generally, a SCC is a set of states in a finite state machine. According to the definition of a SCC, all members of a SCC are able to reach each other through transitions among them. It is intuitive that loops are constructed from transition within every SCC of given services. We call these transitions *Looped Transitions* and build an algorithm for locating them using Tarjan's algorithm of locating SCCs. The algorithm is shown in Algorithm 1 where the sub process is modified from Tarjan's algorithm by adding the part of building the set of *Looped Transitions* Δ^{loop} when a SCC is found (line 16 to 18). Therefore, we can now locate *Unbounded Messages* by gathering the labels in Δ^{loop}.

Algorithm 1: Locating Loop Involved Transitions

Input: an IA4AD $P = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$
Output: set of loop involved transitions: Δ^{loop}

```
1  Procedure SCC(q)
2    begin
3      Lowlink(q) := Number(q) := index := index + 1;
4      push_stack(D, q);
5      foreach (q, a, q') ∈ Δ do
6        if Number(q') is not defined then
7          SCC(q');
8          Lowlink(q) := min(Lowlink(q), Lowlink(q'));
9        else if Number(q') < Number(q) then
10         if on_stack(D, q') then
11           Lowlink(q) := min(Lowlink(q), Number(q'));
12     if Lowlink(q) = Number(q) then
13       while q' = top_stack(D), Number(q') ≥ Number(q) do
14         pop_stack(D);
15         Q^scc ⟵ Q^scc ∪ {q'};
16       foreach δ = (q, a, q') ∈ Δ, a ∈ A^I ∪ A^O do
17         if q, q' ∈ Q^scc then
18           Δ^loop ⟵ Δ^loop ∪ {δ};
19       Q^scc ⟵ ∅;

20 Δ^loop ⟵ ∅; Q^scc ⟵ ∅;
21 empty_stack(D);
22 index := 0;
23 foreach q ∈ Q − {q^0, q^f} do
24   if Number(q) is not defined then
25     SCC(q);

26 return Δ^loop
```

Example 15 and Example 16 demonstrate the algorithm of locating Unbounded Mes-

sages for the FMUS service and the SR service. Note that a SCC may consist only one state. According to the Tarjan's algorithm, if there is no other states which can form a SCC with a certain state, the state itself is marked and returned as a SCC. Also note that for an one state SCC have self transitions, these self transitions are Looped Transitions since a self transition connect a state to the state itself. On the other hand, if an one state SCC has no self transitions, there is no Loop Transition in the SCC. In this case, the one state SCC is ignored.

Example 15 (Unbounded Messages of FMUS service) *For the FMUS service show in Fig. 3.5 where the three services are represented in IA4ADs shown in Example 8, we can locate SCCs of the three services. Note that SCC_i represents set of SCCs where a SCC is represented by a set of states. Also, Δ_i^{loop} represents the set of looped transitions; A_i^{UB} represents the unbounded messages.*

Online Stock Broker*:*

$$SCC_1 = \{\ \{q0_1\},\ \{q1_1\},\ \{q2_1\},\ \{q3_1\},\ \{q4_1\}\ \}$$
$$\Delta_1^{loop} = \{\ (q1_1,\ !RawData,\ q1_1)\ \}$$
$$A_1^{UB} = \{\ RawData\ \}$$

Research Department*:*

$$SCC_2 = \{\ \{q0_2\},\ \{q1_2\},\ \{q2_2, q3_2\},\ \{q4_2\},\ \{q5_2\}\ \}$$
$$\Delta_2^{loop} = \{\ (q2_2,\ ?RawData,\ q3_2),\ (q3_2,\ !Data,\ q2_2)\ \}$$
$$A_2^{UB} = \{\ RawData,\ Data\ \}$$

Online Stock Broker*:*

$$SCC_3 = \{\ \{q0_3\},\ \{q1_3\},\ \{q2_3\},\ \{q3_3\},\ \{q4_3\}\ \}$$
$$\Delta_3^{loop} = \{\ (q2_3,\ ?Data,\ q2_3)\ \}$$
$$A_3^{UB} = \{\ Data\ \}$$

Example 16 (Unbounded Messages of SR service) *For the SR service show in Fig. 5.1 where the two services are represented in IA4ADs shown in Example 13, we can locate SCCs of the two services. Note that SCC_i represents set of SCCs where a SCC is represented by a set of states. Also, Δ_i^{loop} represents the set of looped transitions; A_i^{UB} represents the unbounded messages.*

Sender*:*

$$SCC_1 = \{\ \{q0_1\},\ \{q1_1\},\ \{q2_1\}\ \}$$
$$\Delta_1^{loop} = \{\ (q1_1,\ !c,\ q1_1)\ \}$$
$$A_1^{UB} = \{\ c\ \}$$

Receiver*:*

$$SCC_2 = \{\ \{q0_2\},\ \{q1_2\},\ \{q2_2\}\ \}$$
$$\Delta_2^{loop} = \{\ (q1_2,\ ?c,\ q1_2)\ \}$$
$$A_2^{UB} = \{\ c\ \}$$

5.5 Pushdown Model Checking

To filter out the behavior that avoids behavior mismatches in the synchronous composition of services with Coordinator, the over-behavioral system behavior, pushdown model checking [13] is introduced in the approach. The idea of using pushdown model checking in adaptor generation is similar to detection of behavior mismatches using model checking described in Section 4.2. This time we need the property of Behavior Mismatch Free for the IPS composed by components and Coordinator. This is already described in Section 5.3 and a LTL formula represent the property of adapted Behavior Mismatch Free can be generated for given components.

However, in adaptor generation of the approach, pushdown model checking is used to give us a counterexample which indicates an execution trace where no behavior mismatch is encountered. This can be simply achieved by model checking the *negation* of the property of adapted Behavior Mismatch Free. Since the system behavior composed by the given components and their Coordinator is an over-behavioral IPS as described in Section 5.2. This over-behavioral IPS consists of both good and bad behavior. The counterexample we want is a trace from the good behavior of the IPS. As long as we can specify a property ϕ necessary for adaptors, and the over-behavioral IPS does contain good behavior, the counterexample we want is guaranteed to be returned by pushdown model checking for the *negation* of the property. This trace is considered a candidate behavior of adaptors for given behavioral mismatching components and an adaptor can then be generated from the counterexample. Note that we do not directly use the property of adapted Behavior Mismatch Free for representing the property ϕ necessary for adaptors. This is simply because only the property of adapted Behavior Mismatch Free is not enough for representing suitable property for adaptors. Actually we also need the property of Unbounded Messages mentioned in Section 5.4. We may use the SR service to demonstrate the reason why the property of Behavior Mismatch Free is not enough.

Now we discuss about how to perform pushdown model checking. In this work, pushdown model checking is performed using MOPED [14]. MOPED is a pushdown model checker which can accept directly encoded pushdown system model and perform model checking for a property represented by a LTL formula. However, the propositions in LTL formulas have to be defined using only states or stack symbols. The usage of MOPED is demonstrated in Section 2.5.3. For convenience and avoiding ambiguities, when mentioned in the rest of this thesis, pushdown model checking for a given property refers to pushdown model checking the *negation* of the given property even the word *negation* is not explicitly mentioned. Also, a counterexample returned by pushdown model checking or MOPED should be recognized as the result of pushdown model checking for the *negation* of given property.

For the SR service shown in Fig. 5.1, we can encode the IPS composed from the two services with their Coordinator shown in Example 13 as a pushdown system model for MOPED. The pushdown system model is shown in Fig. 5.3 which include two parts. The first part is the initial configuration which is the initial states coupled with the initial stack symbol z. The second part lists transition relations of the pushdown system. Note when encoding pushing transitions like $(q,\ \epsilon) \hookrightarrow (q',\ < a >)$ where a can be arbitrary message of the pushdown system, we have to concretely specify all possible messages as the stack head symbol a. This means the pushing transition $(q,\ \epsilon) \hookrightarrow (q',\ < a >)$ is encoded as a group of transitions and the number of transitions is the size of the set of

stack symbols, i.e., messages of the system. Actually, the number of transitions built by the encoding should also consider the case when z is the stack head symbol. This makes the number of transitions built from encoding becomes the size of stack symbols plus one. For example, the SR service has three messages so for every pushing transition there are four transitions that using `a`, `b`, `c`, and `z` as the stack head symbols separately. For a popping transition like $(q, \ < a >) \hookrightarrow (q', \ \epsilon)$, the encoded transition relation is one to one mapping since in the case of popping a message, the popped message has to be the stack symbol in the stack head. To demonstrate, Fig. 5.3 shows the pushdown system model for MOPED which is encoded from the IPS computed by adapted synchronous composition of the two services and their Coordinator in Example 13. Composite states are encoded as states with indexes indicates corresponding states of the two services and Coordinator. Transition rules of pushing transitions are grouped to make it easy to compare with transitions computed in Example 13. For example, the first transition of the IPS $(q0_1, q0_2, q0_C), \epsilon) \hookrightarrow ((q1_1, q0_2, q0_C), < a >)$ is encoded as the first four transition rules where the first composite state $(q0_1, q0_2, q0_C)$ is encoded as state `q0_0_c0`. It should be noted that the last transition rule `q2_2_c0 <z> --> q2_2_c0 <z>` is a self transition connecting the final state `q2_2_c0` with the empty stack symbol `z`. This is because algorithms of MOPED require the input pushdown system model behavior continuously or the LTL pushdown model checking will fail even correct property is specified.

We may now build the property of adapted Behavioral Mismatch Free defined in Def. 25 as follows:

```
<> ( q2_2_c0 && z )
```

In this formula, `q2_2_c0` is the final state of the IPS and `z` is the empty stack symbol. Now we may apply pushdown model checking using MOPED for the *negation* of the property of adapted Behavior Mismatch Free given above. The LTL formula has to be converted to a *Never Claim* which is a Büchi automaton equal to the formula. This can be done by SPIN providing the functionality of translating a LTL formula to a Never Claim. Note that a Never Claim usually is translated from the *negation* of specified property so that returned counterexample shows the trace violating the specified property. In the approach, since we perform pushdown model checking for the *negation* of property ϕ, we directly translate ϕ into a Never Claim. Then MOPED reads both the pushdown system model shown in Fig. 5.3 and the Never Claim converted from the LTL formula and perform pushdown model checking. The result of pushdown model checking is a counterexample shown in Fig. 5.4. By observing the trace, we may confirm that the trace correctly reorders messages `a` and `b` so behavior mismatches in SR service are not encountered following this trace.

However, by carefully examining the counterexample, it is disappointing that message `c` is not shown in the trace. Thus, from this trace we can only figure out and build an adaptor with behavior $?a \ ?b \ !b \ !a$. This behavior is not correct comparing to Fig. 5.1. The expected adaptor for SR service should be $?a \ ?c^n \ ?b \ !b \ !c^n \ ?a$ as mentioned in Section 5.1. Therefore, the property of adapted Behavior Mismatch Free is clearly not enough for generating desired adaptors. We need to combine other properties to build the property ϕ necessary for adaptors. The property ϕ should make the message `c` appear in the counterexample. Recall that the message `c` is an Unbounded Message of the SR service

```
#initial state
(q0_0_c0 <z>)

#transition rules
q0_0_c0 <a> --> q1_0_c0 <a a>   q0_0_c0 <c> --> q1_0_c0 <a c>
q0_0_c0 <b> --> q1_0_c0 <a b>   q0_0_c0 <z> --> q1_0_c0 <a z>
q0_0_c0 <b> --> q0_1_c0 <>
q0_1_c0 <a> --> q1_1_c0 <a a>   q0_1_c0 <c> --> q1_1_c0 <a c>
q0_1_c0 <b> --> q1_1_c0 <a b>   q0_1_c0 <z> --> q1_1_c0 <a z>
q0_1_c0 <c> --> q0_1_c0 <>
q0_1_c0 <a> --> q0_2_c0 <>
q0_2_c0 <a> --> q1_2_c0 <a a>   q0_2_c0 <c> --> q1_2_c0 <a c>
q0_2_c0 <b> --> q1_2_c0 <a b>   q0_2_c0 <z> --> q1_2_c0 <a z>
q1_0_c0 <a> --> q1_0_c0 <c a>   q1_0_c0 <c> --> q1_0_c0 <c c>
q1_0_c0 <b> --> q1_0_c0 <c b>   q1_0_c0 <z> --> q1_0_c0 <c z>
q1_0_c0 <a> --> q2_0_c0 <b a>   q1_0_c0 <c> --> q2_0_c0 <b c>
q1_0_c0 <b> --> q2_0_c0 <b b>   q1_0_c0 <z> --> q2_0_c0 <b z>
q1_0_c0 <b> --> q1_1_c0 <>
q1_1_c0 <a> --> q1_1_c0 <c a>   q1_1_c0 <c> --> q1_1_c0 <c c>
q1_1_c0 <b> --> q1_1_c0 <c b>   q1_1_c0 <z> --> q1_1_c0 <c z>
q1_1_c0 <a> --> q2_1_c0 <b a>   q1_1_c0 <c> --> q2_1_c0 <b c>
q1_1_c0 <b> --> q2_1_c0 <b b>   q1_1_c0 <z> --> q2_1_c0 <b z>
q1_1_c0 <c> --> q1_1_c0 <>
q1_1_c0 <a> --> q1_2_c0 <>
q1_2_c0 <a> --> q1_2_c0 <c a>   q1_2_c0 <c> --> q1_2_c0 <c c>
q1_2_c0 <b> --> q1_2_c0 <c b>   q1_2_c0 <z> --> q1_2_c0 <c z>
q1_2_c0 <a> --> q2_2_c0 <b a>   q1_2_c0 <c> --> q2_2_c0 <b c>
q1_2_c0 <b> --> q2_2_c0 <b b>   q1_2_c0 <z> --> q2_2_c0 <b z>
q2_0_c0 <b> --> q2_1_c0 <>
q2_1_c0 <c> --> q2_1_c0 <>
q2_1_c0 <a> --> q2_2_c0 <>
#self-loop for final state
q2_2_c0 <z> --> q2_2_c0 <z>
```

Figure 5.3: SR service: pushdown system model for MOPED

```
--- START ---
q0_0_c0 <z>
q1_0_c0 <a z>
q2_0_c0 <b a z>
q2_1_c0 <a z>
q2_2_c0 <z>
q2_2_c0 <z>

--- LOOP ---
q2_2_c0 <z>
```

Figure 5.4: SR service: counterexample for adapted Behavior Mismatch Free

described in Example 16. Thus, we should now consider the property of *Unbounded Messages* introduced in Section 5.4.

For the purpose of building a LTL property that ensure the appearance of *Unbounded Messages* in the counterexample returned from MOPED, *Looped Transitions* obtained from Algorithm 1 are useful. It is intuitive that a fairness property for all the *Looped Transitions* found in given components should ensure the returned counterexample from pushdown model checking is a trace that goes through all the Looped Transitions at least once. If the returned counterexample is guaranteed traveling through all Looped Transitions, loops in the trace are then established and the arbitrary natural number n indicating Unbounded Messages will also appear. Thus, our purpose of generating suitable adaptors with non-regular behavior can be fulfilled. We call this fairness property the property for *Unbounded Messages* or the property for *Looped Transitions*. Practically, in the set of *Looped Transitions*, every input transition should have corresponding output transition(s) and vice versa. A Looped Transition without corresponding Looped Transition to be synchronized is considered no use in the system and should be deleted from the set.

As mentioned before, MOPED can only accept LTL formulas in which propositions are expressed using states or stack symbols in the input pushdown system model. Therefor, it is impossible to express the property for Looped Transitions since propositions for transition rules are not allowed in MOPED. To solve this situation, special stack symbols for marking Looped Transitions are introduced. Transition rules related to Looped Transitions are modified to insert corresponding special stack symbols. Thus, we can still specify propositions for Looped Transitions through these special stack symbols. For the SR service, the Looped Transitions obtained in Example 15 is the union of Δ_1^{loop} and Δ_2^{loop}. The set of Looped Transitions of the SR service is as follows:

$$\Delta_{SR}^{loop} \ = \ \{ \ (q1_1, \ !c, \ q1_1), \ (q1_2, \ ?c, \ q1_2), \ \}$$

According to the above two Looped Transitions of the SR service, we can build the property for *Looped Transitions*, i.e., property for *Unbounded Messages*, for MOPED as follows:

```
( <> push_c ) && ( <> pop_c )
```

Note that push_c and pop_c are special stack symbols for Looped Transitions Δ_{SR}^{loop}. push_c corresponds to the pop transition ($q1_1$, $!c$, $q1_1$). pop_c corresponds to the push transition ($q1_2$, $?c$, $q1_2$). Using the two special stack symbols, we modify the corresponding transition rules in the pushdown system model for MOPED shown in Fig. 5.3. The modified pushdown system model is shown in Fig. 5.5. The pattern of modification of transition rule is as follows:

- For a push transition rule `s1 <a> --> s2 <b a>` which pushes a symbol `b`, the transition is modified as two transition rules: `s1 <a> --> s2 <push_b a>` and `s2 <push_b> --> s2 <b>` where `push_b` is a special stack symbol.
- For a pop transition rule `s1 <a> --> s2 <>` which pops a symbol `a`, the transition is modified as two transition rules: `s1 <a> --> s1 <pop_a>` and `s1 <pop_a> --> s2 <>` where `pop_a` is a special stack symbol.

For example, transition rule `q1_0_c0 <a> --> q1_0_c0 <c a>` in Fig. 5.3 is modified as two transition rules `q1_0_c0 <a> --> q1_0_c0 <push_c a>` and `q1_0_c0 <c> --> q1_0_c0 <push_c c>`. It should be noted that for the SR service, not every transition rules pushing or popping message `c` is needed to be modified. Composite states in both sides of a transition rule is checked to confirm whether the modification is needed.

Now we can define the property ϕ necessary for adaptors: the conjunction of the property of adapted Behavior Mismatch Free and the property for Looped Transitions. For the SR service, ϕ is as follows:

```
( <> ( q2_2_c0 && z ) ) && ( <> push_c ) && ( <> pop_c )
```

The counterexample returned by pushdown model checking for the *negation* of the property ϕ using MOPED is shown Fig. 5.6. If a configuration has the special symbols push_c or pop_c in its stack, this configuration is no use and can be ignored. By examining this trace, we can confirm that this time the Unbounded Message `c` appears in the counterexample. Therefore, the counterexample is correct for the SR service and a suitable adaptor for the SR service can be generated from this trace.

5.6 Adaptor Generation from Counterexample

In Section 5.5, the process of obtaining a counterexample by pushdown model checking is introduced. The obtained counterexample is a trace representing good behavior in the over-behavioral system behavior composed from given components and their Coordinator. The so called good behavior means the behavior whose execution avoids behavior mismatches of given components as well as make sure the Unbounded Messages of given components appear. This is achieved by pushdown model checking for the property ϕ necessary for adaptors of given components where ϕ consists of two properties: the property of Behavior Mismatch Free and the property for Unbounded Messages or Looped Transitions. In this section, we will show how to generate the behavior interface of an adaptor, i.e., an IPS, from a counterexample returned by pushdown model checking.

```
#initial state
(q0_0_c0 <z>)

#transition rules
q0_0_c0 <a> --> q1_0_c0 <a a>        q0_0_c0 <c> --> q1_0_c0 <a c>
q0_0_c0 <b> --> q1_0_c0 <a b>        q0_0_c0 <z> --> q1_0_c0 <a z>
q0_0_c0 <b> --> q0_1_c0 <>
q0_1_c0 <a> --> q1_1_c0 <a a>        q0_1_c0 <c> --> q1_1_c0 <a c>
q0_1_c0 <b> --> q1_1_c0 <a b>        q0_1_c0 <z> --> q1_1_c0 <a z>
q0_1_c0 <c> --> q0_1_c0 <pop_c>
q0_1_c0 <pop_c> --> q0_1_c0 <>
q0_1_c0 <a> --> q0_2_c0 <>
q0_2_c0 <a> --> q1_2_c0 <a a>        q0_2_c0 <c> --> q1_2_c0 <a c>
q0_2_c0 <b> --> q1_2_c0 <a b>        q0_2_c0 <z> --> q1_2_c0 <a z>
q1_0_c0 <a> --> q1_0_c0 <push_c a>   q1_0_c0 <c> --> q1_0_c0 <push_c c>
q1_0_c0 <b> --> q1_0_c0 <push_c b>   q1_0_c0 <z> --> q1_0_c0 <push_c z>
q1_0_c0 <push_c> --> q1_0_c0 <c>
q1_0_c0 <a> --> q2_0_c0 <b a>        q1_0_c0 <c> --> q2_0_c0 <b c>
q1_0_c0 <b> --> q2_0_c0 <b b>        q1_0_c0 <z> --> q2_0_c0 <b z>
q1_0_c0 <b> --> q1_1_c0 <>
q1_1_c0 <a> --> q1_1_c0 <push_c a>   q1_1_c0 <c> --> q1_1_c0 <push_c c>
q1_1_c0 <b> --> q1_1_c0 <push_c b>   q1_1_c0 <z> --> q1_1_c0 <push_c z>
q1_1_c0 <push_c> --> q1_1_c0 <c>
q1_1_c0 <a> --> q2_1_c0 <b a>        q1_1_c0 <c> --> q2_1_c0 <b c>
q1_1_c0 <b> --> q2_1_c0 <b b>        q1_1_c0 <z> --> q2_1_c0 <b z>
q1_1_c0 <c> --> q1_1_c0 <pop_c>
q1_1_c0 <pop_c> --> q1_1_c0 <>
q1_1_c0 <a> --> q1_2_c0 <>
q1_2_c0 <a> --> q1_2_c0 <push_c a>   q1_2_c0 <c> --> q1_2_c0 <push_c c>
q1_2_c0 <b> --> q1_2_c0 <push_c b>   q1_2_c0 <z> --> q1_2_c0 <push_c z>
q1_2_c0 <push_c> --> q1_2_c0 <c>
q1_2_c0 <a> --> q2_2_c0 <b a>        q1_2_c0 <c> --> q2_2_c0 <b c>
q1_2_c0 <b> --> q2_2_c0 <b b>        q1_2_c0 <z> --> q2_2_c0 <b z>
q2_0_c0 <b> --> q2_1_c0 <>
q2_1_c0 <c> --> q2_1_c0 <pop_c>
q2_1_c0 <pop_c> --> q2_1_c0 <>
q2_1_c0 <a> --> q2_2_c0 <>
#self-transition from final state(s)
q2_2_c0 <z> --> q2_2_c0 <z>
```

Figure 5.5: SR service: pushdown system model for MOPED with special stack symbols

```
--- START ---
q0_0_c0 <z>
q1_0_c0 <a z>
q1_0_c0 <push_c a z>
q1_0_c0 <c a z>
q1_0_c0 <push_c c a z>
q1_0_c0 <c c a z>
q2_0_c0 <b c c a z>
q2_1_c0 <c c a z>
q2_1_c0 <pop_c c a z>
q2_1_c0 <c a z>
q2_1_c0 <pop_c a z>
q2_1_c0 <a z>
q2_2_c0 <z>
q2_2_c0 <z>

--- LOOP ---
q2_2_c0 <z>
```

Figure 5.6: SR service: counterexample for adapted Behavior Mismatch Free and Unbounded Messages

Generally, a counterexample is a trace of execution represented by a sequence of configurations. A configuration is a pair consists of a composite state and the corresponding stack contents represented by a sequence of stack symbols. In the approach, we provide an algorithm which directly converts the sequence of configurations into the behavior interface of an adaptor represented by an IPS. The idea of adaptor generation from a counterexample is quite simple. First, for each configuration in the counterexample, the composite state could be taken as a state of an IPS, i.e., the adaptor we want to generate. Then transitions are generated to connect these states following the order of the sequence of configurations. Recall there are three kinds of transitions in an IPS: push, pop, and internal. Generally, we generation transitions by checking the corresponding stack contents to figure out which kind of transition should be generated and then generate the transitions with information in corresponding configurations.

For more detailed explanation, Algorithm 2 shows the algorithm for generating an adaptor from a counterexample obtained from pushdown model checking using MOPED for *negation* of the property ϕ. In the algorithm, the input is a finite sequence of configurations which are pairs of states and stack contents. Configurations in the counterexample are denoted by $c_i = (s_i, w_i)$, $i \in [0, m]$. The natural number m indicates the number of configurations. Note that the last input is a natural number k indicates the loop start index of the counterexample. The natural number k is important because a counterexample returned by model checkers is actually expressed by two sequences of configurations and the second one shows a looped trace of the counterexample. Thus, a counterexample is actually an ω infinite trace

$$c_0 c_1 c_2 \dots c_{k-1} \ (c_k c_{k+1} \dots c_m)^\omega$$

According to the above sequence of configurations, the configuration next to the last configuration c_m is c_k which is the configuration starting the looped trace. Now we may explain the details in Algorithm 2 as follows:

Algorithm 2: Counterexample to adaptor

Input: A set of composable IA4AD: $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$, $i \in [1, n]$; Configurations $c_i = (s_i, w_i)$, $i \in [0, m]$; Loop start index k.

Output: Adaptor $D = (Q_D, q_D^0, \Gamma_D, z, T_D, F_D)$

1 $Q_D \longleftarrow \{s_k\}$; $q_D^0 \longleftarrow s_0$; $T \longleftarrow \emptyset$;
2 $\Gamma_D := \bigcup_i A_i^O \cup \{z\} \cup \{\epsilon\}$;
3 **foreach** $c_i = (s_i, w_i)$, $i \in [0, m]$ **do**
4 **if** $i = m$ **then**
5 $(s_i', w_i') = (s_k, w_k)$
6 **else**
7 $(s_i', w_i') = (s_{i+1}, w_{i+1})$
8 $Q_D \longleftarrow Q_D \cup \{s_i\}$;
9 **if** $|w_i| - |w_i'| = 1$ **then**
10 $T_D \longleftarrow T_D \cup \{(s_i, w_i(0)) \hookrightarrow (s_i', \epsilon)\}$;
11 **if** $|w_i| - |w_i'| = -1$ **then**
12 $T_D \longleftarrow T_D \cup \{(s_i, w_i(0)) \hookrightarrow (s_i', w_i'(0)w_i(0))\}$;
13 **if** $|w_i| - |w_i'| = 0 \wedge s_i' \neq q_D^0$ **then**
14 $T_D \longleftarrow T_D \cup \{(s_i, w_i(0)) \hookrightarrow (s_i', w_i'(0)),\ (s_i', w_i'(0)) \hookrightarrow (s_i, w_i(0))\}$;
15 $F_D \longleftarrow Q_D$;
16 **return** $D = (Q_D, q_D^0, \Gamma_D, z, T_D, F_D)$

1. First, we choose a pair of adjacent two configurations $c_i = (s_i, w_i)$ and $c_{i+1} = (s_{i+1}, w_{i+1})$ then do the following actions:
   (a) Put the two states s_1 and s_{i+1} into the set of states of the IPS, i.e., the adaptor (line 8);
   (b) Compare the two stack contents w_i and w_{i+1} for building a transition connecting the above two states (line 9 to 14).
2. Choose another pair of adjacent two configurations and start over until all pairs of adjacent configurations are processed.

In the step of 1-(b), since an IPS only has push, pop, and internal transitions, adjacent stack contents are differed by 1 or equal in length. This means, if length of w_i is longer than of w_{i+1} by 1, the transition connecting s_i and s_{i+1} is a pop transition; if length of w_i is shorter than of w_{i+1} by 1, the transition connecting s_i and s_{i+1} is a push transition. Also, if length of w_i is equal to w_{i+1}, the transition connecting s_i and s_{i+1} is an internal transition. Internal transition are actually corresponding to internal actions of given components and make no communication. Note that the algorithm generates internal transitions

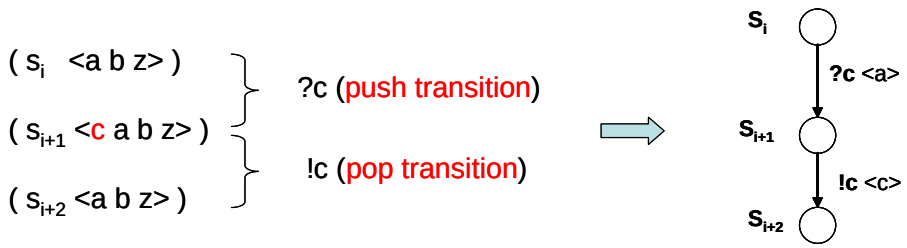


Figure 5.7: Counterexample to Adaptor: illustrating for a segment of counterexample

bidirectional (line 14) for the possibility of indetermination of internal transitions in given components. Also note that we do not generate internal transitions that goes to the initial state (line 13). This violates the *One Session Process* Policy in the approach. The generated IPS, i.e., the adaptor, is then ready to guide the system of components avoiding behavior mismatches and guaranteeing the appearance of *Unbounded Messages*.

For more demonstration, Fig. 5.7 shows a graphical illustration for easier understanding. The left part is a segment of counterexample consists of three configurations. The middle part is the decision from comparison of the stack contents of adjacent two configurations. By Comparing `, <a b z>` and `<c a b z>`, the stack contents of the first and second configurations, we may conclude that s_i and s_{i+1} are connected by a push transition which pushes message `c` into the stack. Similarly, we may conclude that s_{i+1} and s_{i+2} are connected by a pop transition which pops message `c` from the stack. The right part is the segment of an IPS showing the states and translations we generated for the adaptor.

Finally, we would like to give another demonstration on generating an adaptor for the SR example. The demonstration is shown in Example 17.

Example 17 *We follow Algorithm 2 to generate an adaptor from the counterexample generated by MOPED shown in Fig. 5.6. Note that we ignore configurations having special stack symbols, i.e.,* `push_c` *and* `pop_c` *since they are no use in adaptor generation. The generated adaptor* $D = (Q_D, q_D^0, \Gamma, z, T_D, F_D)$ *is shown as follows:*

- $Q_D = \{\ q0_D,\ q1_D,\ q2_D,\ q3_D,\ q4_D\ \}$.
- $q_D^0 = q0_D$.
- $\Gamma = \{\ a,\ b,\ c\ \} \cup \{\ z\ \}$.
- $z \in \Gamma$ *is initial symbol of stack.*
- $T_C = \{\ (q0_D,\ z) \hookrightarrow (q1_D,\ \ < a >),$
  $(q1_D,\ a) \hookrightarrow (q1_D,\ \ < c\ a >),$
  $(q1_D,\ a) \hookrightarrow (q2_D,\ \ < b\ a >),$
  $(q1_D,\ c) \hookrightarrow (q2_D,\ \ < b\ c >),$
  $(q2_D,\ b) \hookrightarrow (q3_D,\ \epsilon),$
  $(q3_D,\ c) \hookrightarrow (q3_D,\ \epsilon),$
  $(q3_D,\ a) \hookrightarrow (q4_D,\ \epsilon)\ \}$
- $F_D = Q_D$.

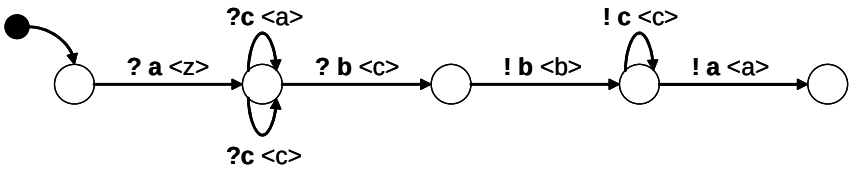


Figure 5.8: SR service: the generated adaptor

Note that `q0_D` *to q4_D represent composite states* `q_0_0_c0`, `q_1_0_c0`, `q_2_0_c0`, `q_2_1_c0`, *and* `q_2_2_c0`. *The generated adaptor is drawn in Fig. 5.8. Also note that transition label* `?c <a>` *means a pushing transition that pushes message* `c` *into stack while the stack head symbol is* `a`; *transition label* `!c <c>` *means a popping transition that pops message* `c` *from stack head while the stack head symbol is* `c`.

Chapter 6

Tool Implementation

6.1 Overview

For conducting experiments with the approach, a tool is implemented. The architecture of the tool is shown in Fig. 6.1. As the architecture shows, the tool has two parts. The first part reads behavior interfaces of components and generates a pushdown system and LTL properties for MOPED. The second part then reads the counterexample produced by MOPED after pushdown model checking and generate an adaptor using Algorithm 2. In general, after reading the input, i.e., behavior interfaces of components, following steps are processed:

1. Perform compatibility check.
2. Output the Promela model along with the LTL formula of behavior mismatch free for SPIN.
3. Locate looped transitions in all services.
4. Compute adapted synchronous composition of services with coordinator and output the pushdown system model for MOPED.
5. Output the LTL formula of property of behavior mismatch free and fairness property of looped and branching transitions for MOPED.
6. After a counterexample is generated by MOPED, read the counterexample and generate an IPS, i.e. the adaptor of given services.

Note that in the architecture, the part of detection of behavior mismtach using SPIN is not shown. This means in the tool, the step of detection of behavior mismatch free can be omitted if existence of behavior mismatches is already confirmed. In the following sections, we will describe details of the tool.

6.2 Read Input

First of all, behavior interfaces of components are encoded as input of the tool. The input file specifies behavior interfaces of components with the following information: a) initial

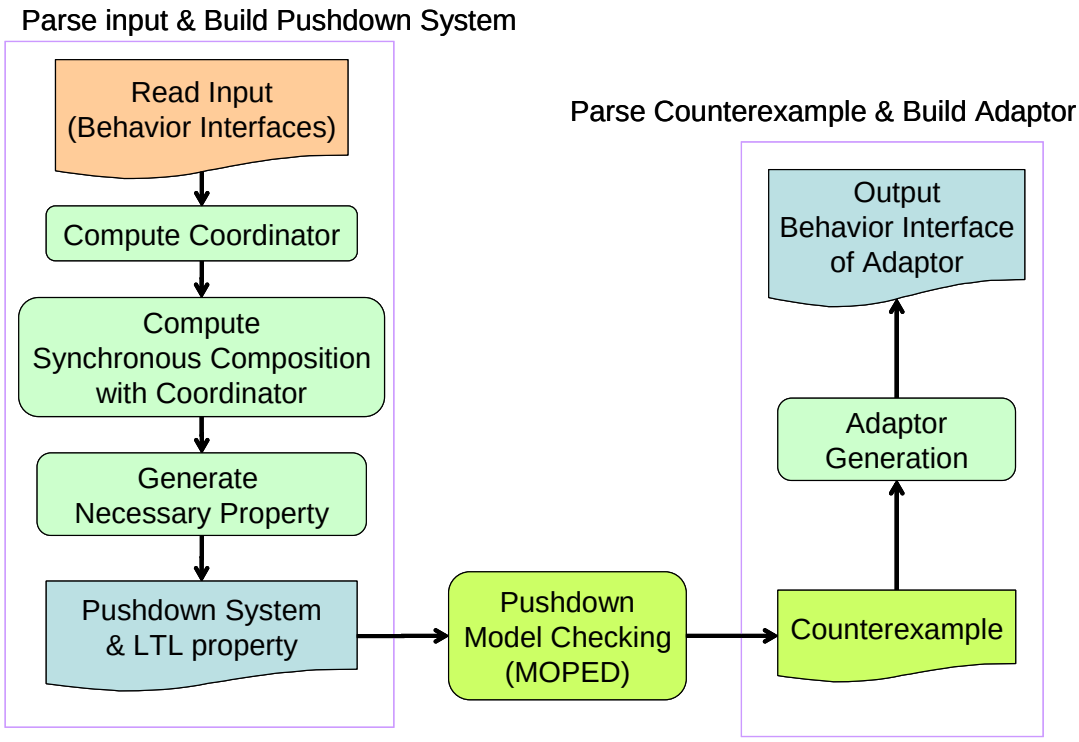


Figure 6.1: Tool Architecture

state; b) final state; c) transitions having labels with prefix of "!" or "?" or no prefix. The grammar of the input is shown in BNF as follows.

```
defs
  : service defs
  | service

component
  : START_DEF service_name INIT init_state FINAL final_state transitions

service_name
  : IDENTIFIER

init_state
  : IDENTIFIER

final_state
  : IDENTIFIER

transitions
  : transition transitions
  | transition

transition
```

```
  : '(' state ',' '?' message ',' state ')'
  | '(' state ',' '!' message ',' state ')'
  | '(' state ','     message ',' state ')'

state
  : IDENTIFIER

message
  : IDENTIFIER
```

The input consists of blocks of specifications of behavior interfaces of components. Each component has the following information specified:

- `service_name`: name of the component
- `init_state`: the initial state of the component.
- `final_state`: the final state of the component.
- `transition`: transition rule. `tranisions` shows sequence of transitions rules. A transition rule is expressed in the form of `(s1, a, s2)` where `s1` and `s2` are state names and `a` is the message name with `!` or `?` prefixed.
- `state`: state name.
- `message`: message name.

Note that keywords expressed by capital letters are tokens. `START_DEF` indicats the start of specification of a component and is expressed as `service::` in the input. `INIT` indicats the start of specification of initial state and is expressed as `init::` in the input. `FINAL` indicats the start of specification of initial state and is expressed as `final::` in the input. `IDENTIFIER` indicates a sequence of alphabets and digits which shows names of components, states, and messages. In the input, the input/output/internal alphabets of a components are not explicitly specified. The tool will calculate sets of alphabets from transitions and generate arraies to keep the alphabets. This should save the efforts of developers in encoding behavior interfaces of components in the input for the tool.

After the input is read and parsed, the tool will start the first task: compatibility check. Compatibility check is based the condition of compatibility defined in Def. 14 which includs the following checks:

- Inputs and outputs of a component must be exclusive.
- Different components have exclusive inputs and exclusive outputs.
- Union of inputs and union of outputs are the same set.
- Initial and final states are different states.

The last check is in the definition of condition of compatibility but defined in constraint of IA4AD which requires a component be represented as an IA4AD with different initial and final states. As an example, we use the simple SR service demonstrated in Example 13. The behavior interfaces of the two services in SR service is encoded as input of the tools as follows:

```
# Sender Receiver Example

service:: Sender
init::S0
final::S2
(S0,!a,S1)
(S1,!c,S1)
(S1,!b,S2)

service:: Receiver
init::S0
final::S2
(S0,?b,S1)
(S1,?c,S1)
(S1,?a,S2)
```

The line started with # is a comment line and is ignored while read by the tool. In the input, we may use same state names in different components without causing problems in computation. The tool may display a result of parsing the input which shows contents of each components including states names, messages, and the result of compatibility check. The result of parsing is shown as follows:

```
project_name  : sr
num_service   : 2
num_message_io: 3
num_message_h : 0
---messages_IO[0]: a
---messages_IO[1]: c
---messages_IO[2]: b

service name: Sender
num_state: 3
---states[0]: S0
---states[1]: S1
---states[2]: S2
init_state     : 0
final_state    : 2
num_T_i : 0
num_T_o : 3
num_T_h : 0
```

```
T_o[0] : ( 0, 0, 1 )
T_o[1] : ( 1, 1, 1 )
T_o[2] : ( 1, 2, 2 )

service name: Receiver
num_state: 3
---states[0]: S0
---states[1]: S1
---states[2]: S2
init_state     : 0
final_state    : 2
num_T_i : 3
num_T_o : 0
num_T_h : 0
T_i[0] : ( 0, 2, 1 )
T_i[1] : ( 1, 1, 1 )
T_i[2] : ( 1, 0, 2 )

compatibility check OK!!

initial and final states check OK!!
```

By reviewing the result of parsing, developers can confirm if components are correctly specified. For example, if the number of messages are wrong or compatibility check is failed, there might be typos in names of messages. Also, one may notice that all names of messages and states are stores as integer numbers which can save execution time in computation such as comparing message names for synchronization.

6.3 Promela Model for SPIN

After parsing the input, Promela model is generated by the tool for detection of behavior mismatches. In Promela model, components are intuitively encoded as automata communicating through a synchronous channel. For example, Promela model for the SR service is demonstrated below:

```
#define accept_Sender (Sender:active_state==2)
#define accept_Receiver (Receiver:active_state==2)

/* define names of messages */
mtype = {a, c, b};

/* one synchronous channel for all services */
chan sync_chan = [0] of { mtype };

/* interface protocol of Sender */
proctype Sender(chan q) {
```

```
short active_state;
active_state = 0; /* initial state */
do
:: (active_state == 0) ->
    if
    :: q!a -> active_state = 1
    fi;
:: (active_state == 1) ->
    if
    :: q!c -> active_state = 1
    :: q!b -> active_state = 2
    fi;
od
}

/* interface protocol of Receiver */
proctype Receiver(chan q) {
short active_state;
active_state = 0; /* initial state */
do
:: (active_state == 0) ->
    if
    :: q?b -> active_state = 1
    fi;
:: (active_state == 1) ->
    if
    :: q?c -> active_state = 1
    :: q?a -> active_state = 2
    fi;
od
}

/* init process */
init {
  run Sender(sync_chan);
  run Receiver(sync_chan);
}
```

The structure is same as the Promela model for the FMUS service demonstrated in Section 4.2. The first two lines define the propositions of acceptance condition of the two services. The two propositions are also used to define the LTL formula of Behavior Mismatch Free as follows:

```
<> ( accept_Sender && accept_Receiver )
```

Behavior of components are defined as processes in Promela separately within the block marked by `do` and `od`. Processes in Promla model are communicating through a

synchronous channel `q` as declared after keyword `proctype` with parameter `(chan q)`. Then we can perform detection of behavior mismatches by model checking using SPIN.

6.4 Pushdown Systems Model for MOPED

Note that since MOPED only accepts a pushdown system model as input, the tool has to generate the Coordinator for input components and then compute the adapted synchronous composition of the components and their Coordinator. The tool then output the composed IPS as a pushdown system model for MOPED. The tool also prepares a LTL formula which is the conjunction of the property of adapted Behavior Mismatch Free and the property for Unbounded Messages / Looper Transitions. As mentioned in Section 5.4, Looped Transitions are obtained following Algorithm 1 through finding SCCs using the Tarjan's algorithm. Then pushdown model checking is performed to check the pushdown system model for the LTL property.

The pushdown system model for the SR service is already shown in Fig. 5.5 and explanations about the pushdown system model are already given in Section 5.5. Therefore, demonstration on the SR service is skipped here. However, we would like to address again that since MOPED does not provide functionality of labeling transitions but only states and stack symbols can be used as propositions, modifications on transition rules are necessary for expressing fairness property of Looped Transitions. For a push transition rule, for example `s1 <a> --> s2 <b a>` which pushes a symbol `b`, the transition is modified as two transitions: `s1 <a> --> s2 <push_b a>` and `s2 <push_b> --> s2 <b>`. Similarly, for a pop transition , for example `s1 <a> --> s2 <>` which pops a symbol `a`, the transition is modified as two transitions: `s1 <a> --> s1 <pop_a>` and `s1 <pop_a> --> s2 <>`. The two symbols `push_b` and `pop_a` are then specified as atomic propositions in the fairness property.

6.5 Adaptor Generation from Counterexample

When MOPED returned a counterexample after pushdown model checking, the tool reads the counterexample and generate the adaptor from the counterexample. The computation follows steps in Algorithm 2. First, the tool read the counterexample and parse the trace for adaptor generation. Recall a counterexample is a finite sequence of configurations. The tool parse a counterexample using the following BNF syntax:

```
trace
  : TRACE_START configurations TRACE_LOOP configurations

configurations
  : config configurations
  | config

config
  : state '<' stack_contents '>'

stack_contents
```

```
  : stack_symbol stack_contents
  | stack_symbol

state
  : IDENTIFIER

stack_symbol
  : IDENTIFIER
```

Basically, a counterexample consists of two sequences of configurations. We use tokens `TRACE_START` and `TRACE_LOOP` to express the starting of the two sequences separately. Each sequence of configurations consists of a list of configurations. Each configuration consists of a pair of state and stack contents. A stack contents is a list of stack symbols. Finally, states and stack symbols are marks by token `IDENTIFIER`. For easier understanding, one may refer to the counterexample for the SR service shown in Fig. 5.6 to confirm the above syntax.

After the tool reads and parses the counterexample. A result is generated for confirming correctness of the parsing. For example, the result of parsing the counterexample of the SR service shown in Fig. 5.6 is listed as follows:

```
num_messages                 : 4
num_states                   : 5
num_config[0], num_config[1] : 10, 1

---messages[0]: z
---messages[1]: a
---messages[2]: c
---messages[3]: b

---states[0]: q0_0
---states[1]: q1_0
---states[2]: q2_0
---states[3]: q2_1
---states[4]: q2_2

---config[0][0]:
------state = 0 (q0_0_c0)
------stack_length = 1
---------stack[0] = 0 (z)
---config[0][1]:
------state = 1 (q1_0_c0)
------stack_length = 2
---------stack[0] = 1 (a)
---------stack[1] = 0 (z)
---config[0][2]:
------state = 1 (q1_0_c0)
```

```
------stack_length = 3
---------stack[0] = 2 (c)
---------stack[1] = 1 (a)
---------stack[2] = 0 (z)
---config[0][3]:
------state = 1 (q1_0_c0)
------stack_length = 4
---------stack[0] = 2 (c)
---------stack[1] = 2 (c)
---------stack[2] = 1 (a)
---------stack[3] = 0 (z)
---config[0][4]:
------state = 2 (q2_0_c0)
------stack_length = 5
---------stack[0] = 3 (b)
---------stack[1] = 2 (c)
---------stack[2] = 2 (c)
---------stack[3] = 1 (a)
---------stack[4] = 0 (z)
---config[0][5]:
------state = 3 (q2_1_c0)
------stack_length = 4
---------stack[0] = 2 (c)
---------stack[1] = 2 (c)
---------stack[2] = 1 (a)
---------stack[3] = 0 (z)
---config[0][6]:
------state = 3 (q2_1_c0)
------stack_length = 3
---------stack[0] = 2 (c)
---------stack[1] = 1 (a)
---------stack[2] = 0 (z)
---config[0][7]:
------state = 3 (q2_1_c0)
------stack_length = 2
---------stack[0] = 1 (a)
---------stack[1] = 0 (z)
---config[0][8]:
------state = 4 (q2_2_c0)
------stack_length = 1
---------stack[0] = 0 (z)
---config[0][9]:
------state = 4 (q2_2_c0)
------stack_length = 1
---------stack[0] = 0 (z)
---config[1][0]:
------state = 4 (q2_2_c0)
```

```
------stack_length = 1
---------stack[0] = 0 (z)
```

The result list shows the data gained from the input counterexample. First, there is a summary which lists global information got from parsing: number of messages, number of states, and number of configurations. Note that there are two numbers of configurations in which the second is the looped trace. In the SR service, since we only allow one final states in each component, the is also only one composite final state which is the one itself constructs the looped trace. Then names of messages and states corresponding to their indexes assigned by the tool is listed. Finally, details of read configurations are listed with following data:

- Index of configuration.
- Index and name of the composite state of the configuration.
- Length of the stack of the configuration.
- Stack contents of the configuration which lists the index and name of every stack symbol.

Developers can check the result and see if the parse is correctly done by the tool. Right after the parsing result, a briefing of generated adaptor is also listed. The information of the generated adaptor can then be used for further use such as implementation or drawing a graph for illustration. As an example the adaptor generated from the counterexample of the SR service shown in Fig. 5.6 is listed as follows:

```
---adaptor
------num_state : 5
------num_T_i   : 4
------num_T_o   : 3
------num_T_h   : 0
---------T_i[0] : ( 0 < 0 > --> 1 < 1, 0 > )
---------T_i[1] : ( 1 < 1 > --> 1 < 2, 1 > )
---------T_i[2] : ( 1 < 2 > --> 1 < 2, 2 > )
---------T_i[3] : ( 1 < 2 > --> 2 < 3, 2 > )
---------T_o[0]  : ( 2 < 3 > --> 3 < -1, -1 > )
---------T_o[1]  : ( 3 < 2 > --> 3 < -1, -1 > )
---------T_o[2]  : ( 3 < 1 > --> 4 < -1, -1 > )
```

Note that in above adaptor, zero and positive integers in transitions represent messages while -1 represents empty, i.e., ϵ. For example, the last transition `T_o[0]: (3 <1> --> 4 <-1,-1>)` is a pop transition which pops the message `a` indexed by 1. This transition is actually expressing `q2_1 <a> --> q2_2 <>` in the form of transitions in pushdown system model for MOPED. Though the adaptor is shown using integers, we may refer to the parsing result to confirm the names of messages and states in each transition of

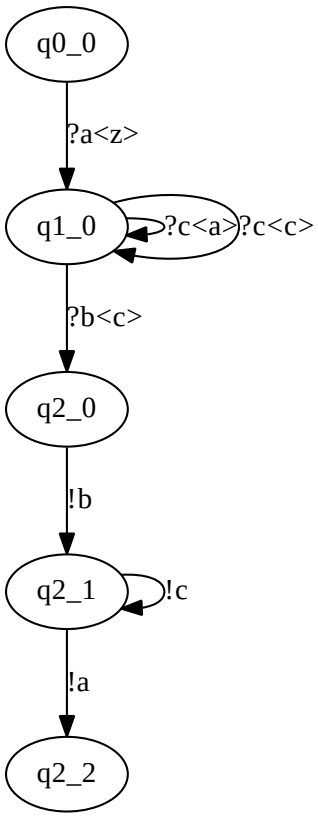


Figure 6.2: SR service: adaptor drawn by Graphviz

the adaptor. We may further compare withe the adaptor of the SR service shown in Example 17 to confirm that the adaptor generated by the tool is the one we are expecting. Also, the adaptor of the SR service generated by the tool listed above can be drawn in graph which is just the one already showed in Fig. 5.8. Also, the tool provides an output for drawing graph of generated adaptor using Graphviz [17] so that we may see the generated adaptor being demonstrated graphically as soon as it is generated. The output for Graphviz for the SR services is shown below and the drawing using Graphviz is shown in Fig. 6.2. Actually the adaptor shown in Fig. 5.8 is based on the graph shown in Fig. 6.2.

```
digraph Adaptor {
q0_0 -> q1_0 [label="?a<z>"];
q1_0 -> q1_0 [label="?c<a>"];
q1_0 -> q1_0 [label="?c<c>"];
q1_0 -> q2_0 [label="?b<c>"];
q2_0 -> q2_1 [label="!b"];
q2_1 -> q2_1 [label="!c"];
q2_1 -> q2_2 [label="!a"];
}
```

Chapter 7

Experiments

This chapter demonstrates some experiments for the approach. All the experiments are conducted using the tool introduced in Chapter 6. The purpose of the experiments is to test the approach proposed in this work on the following aspects:

- **To confirm the solution to the motivational example, the FMUS service**

  The FMUS service is a typical problem with reordering behavior mismatches which need an adaptor with non-regular adaptor. We want to see if the approach can successfully solve this problem.

- **To examine the feasibility of implementing adaptors on real platforms of software development**

  The approach uses pushdown systems model to represent adaptors which is unusual in software design. We want to make sure the modeling in the approach is applicable on real platform of software development. For the FMUS services, we choose the BPEL implementation of web services and give some direction on developing BPEL adaptors.

- **To examine if the approach is applicable on common adaptation problems**

  The motivational example is typical but not representative of common adaptation problems. We want examine other problems of adaptation commonly encountered and see if the approach is still applicable. According to our investigation, adaptation problems in which components have signature mismatches and/or branchings are enumerated.

7.1 Fresh Market Update Service

In this section, the motivational example *Fresh Market Update Service* shown in Fig. 3.5 is subject for the experiments. The behavior interface of the three services to be composed are encoded in an input file shown in Fig. 7.1. For each service, the service name as well as initial and final states are explicitly specified. Then transitions are specified with special symbols "!" and "?" to define input and output transitions respectively. This input file is read and parse by the tool. The parsing result is shown below. We may confirm the parsing is correct and the compatibility check is passed.

```
service:: OnlineStockBroker
init::S0
final::S4
(S0,!RawData,S1) (S1,!RawData,S1) (S1,!EndOfData,S2)
(S2,!Start,S3) (S3,?Ack,S4)

service:: ResearchDepartment
init::S0
final::S5
(S0,?RawData,S1) (S1,!Data,S2) (S2,?RawData,S3)
(S3,!Data,S2) (S2,?EndOfData,S4) (S4,!Complete,S5)

service:: Investor
init::S0
final::S4
(S0,?Start,S1) (S1,?Data,S2) (S2,?Data,S2)
(S2,?Complete,S3) (S3,!Ack,S4)
```

Figure 7.1: FMUS service: input file

```
project_name  : fmus
num_service   : 3
num_message_io: 6
num_message_h : 0
---messages_IO[0]: RawData
---messages_IO[1]: EndOfData
---messages_IO[2]: Start
---messages_IO[3]: Data
---messages_IO[4]: Complete
---messages_IO[5]: Ack

service name: OnlineStockBroker
num_state: 5
---states[0]: S0
---states[1]: S1
---states[2]: S2
---states[3]: S3
---states[4]: S4
init_state     : 0
final_state    : 4
num_T_i : 1
num_T_o : 4
num_T_h : 0
T_i[0] : ( 3, 5, 4 )
```

```
T_o[0] : ( 0, 0, 1 )
T_o[1] : ( 1, 0, 1 )
T_o[2] : ( 1, 1, 2 )
T_o[3] : ( 2, 2, 3 )

service name: ResearchDepartment
num_state: 6
---states[0]: S0
---states[1]: S1
---states[2]: S2
---states[3]: S3
---states[4]: S4
---states[5]: S5
init_state     : 0
final_state    : 5
num_T_i : 3
num_T_o : 3
num_T_h : 0
T_i[0] : ( 0, 0, 1 )
T_i[1] : ( 2, 0, 3 )
T_i[2] : ( 2, 1, 4 )
T_o[0] : ( 1, 3, 2 )
T_o[1] : ( 3, 3, 2 )
T_o[2] : ( 4, 4, 5 )

service name: Investor
num_state: 5
---states[0]: S0
---states[1]: S1
---states[2]: S2
---states[3]: S3
---states[4]: S4
init_state     : 0
final_state    : 4
num_T_i : 4
num_T_o : 1
num_T_h : 0
T_i[0] : ( 0, 2, 1 )
T_i[1] : ( 1, 3, 2 )
T_i[2] : ( 2, 3, 2 )
T_i[3] : ( 2, 4, 3 )
T_o[0] : ( 3, 5, 4 )

compatibility check OK!!
initial and final states check OK!!
```

After parsing the input file, the tool generates Promela model for SPIN to perform detection of behavior mismatches by model checking for the property of Behavior Mismatch Free. This part is already described in detail in Section 4.2 and the existence of behavior mismatches is confirmed in the FMUS service by the approach. Thus, there is no need to explain this part again. We may proceed to the part of pushdown model checking and adaptor generation.

The tool also computes Coordinator for the FMUS service as already shown in Example 12 and generates a pushdown system model for MOPED which partly shown in Fig. 7.2. Recall that the synchronous composition of the three services with their coordinator is an IPS, this pushdown system model is basically encoded from the IPS by the tool. The LTL formula shown in Fig. 7.3 is the conjunction of behavior mismatch free `<>` `( q4_5_4_c0 && z )` and fairness property of Looped Transitions. As already explained in Section 5.5 and Section 6.4, to express the fairness property of Looped Transitions found, modification on transition rules using special stack symbols are introduced to the pushdown system model.

After pushdown model checked by MOPED for the *negation* of the LTL formula, a counterexample returned and shown in Fig. 7.4. As already mentioned in Section 5.5, in order to satisfy fairness property of Looped Transitions, Looped Transitions are traveled in the trace. This can be confirmed by the existence of special symbols shown in stack contents of configurations in the counterexample. The counterexample then is read by the tool and an adaptor represented by IPS is generated. Behavior interface of the generated adaptor returned by the tool is shown in Fig. 7.5. The data can be used to constructed an IPS shown in Fig. 7.6 and draw a graphical illustration showed in Fig. 7.7.

Comparing to the *expected adaptor* of the FMUS service in Example 10, the generated adaptor has the following behavior:

$$?R\ !R\ ?D\ (?R\ !R\ ?D)^{n-1}\ ?R\ !R\ ?E\ ?S\ !S\ ?D\ !D\ !E\ !D^n\ ?C\ !C\ ?A\ !A,\ n > 1$$

It is easy to confirm that in both behavior, all messages received by the two adaptors are finally sent, which satisfies the basic requirement of an adaptor, i.e., the property of adapted Behavior Mismatch Free. Now we look at the part related to *Unbounded Messages*. In the generated adaptor, the Unbounded Messages are emphasized by natural numbers n and $n-1$. This means the Unbounded Messages are successfully captured while recalling that the expected adaptor is designed for the purpose of showing Unbounded Messages. It should be noticed that in the behavior of the generated, n and $n-1$ actually are resulted by the structures of behavior interfaces of the three services where messages `RawData` and `Data` are not all packed in one group but separated. We may conclude that the approach is successful on generating adaptors with non-regular behavior for behavioral mismatching components.

7.2 BPEL implementation

Since the approach uses pushdown system model for representing adaptors which is unusual to general implementation of software components or web services. We should

```
#initial state
(q0_0_0_c0 <z>)

#transition rules
q0_0_0_c0 <RawData> --> q1_0_0_c0 <RawData RawData>
q0_0_0_c0 <EndOfData> --> q1_0_0_c0 <RawData EndOfData>
q0_0_0_c0 <Start> --> q1_0_0_c0 <RawData Start>
q0_0_0_c0 <Data> --> q1_0_0_c0 <RawData Data>
q0_0_0_c0 <Complete> --> q1_0_0_c0 <RawData Complete>
q0_0_0_c0 <Ack> --> q1_0_0_c0 <RawData Ack>
q0_0_0_c0 <z> --> q1_0_0_c0 <RawData z>
q0_0_0_c0 <RawData> --> q0_1_0_c0 <>
q0_0_0_c0 <Start> --> q0_0_1_c0 <>
.....
q1_1_3_c0 <RawData> --> q1_1_3_c0 <push_RawData RawData>
q1_1_3_c0 <EndOfData> --> q1_1_3_c0 <push_RawData EndOfData>
q1_1_3_c0 <Start> --> q1_1_3_c0 <push_RawData Start>
q1_1_3_c0 <Data> --> q1_1_3_c0 <push_RawData Data>
q1_1_3_c0 <Complete> --> q1_1_3_c0 <push_RawData Complete>
q1_1_3_c0 <Ack> --> q1_1_3_c0 <push_RawData Ack>
q1_1_3_c0 <z> --> q1_1_3_c0 <push_RawData z>
q1_1_3_c0 <push_RawData> --> q1_1_3_c0 <RawData>
q1_1_3_c0 <RawData> --> q2_1_3_c0 <EndOfData RawData>
q1_1_3_c0 <EndOfData> --> q2_1_3_c0 <EndOfData EndOfData>
q1_1_3_c0 <Start> --> q2_1_3_c0 <EndOfData Start>
q1_1_3_c0 <Data> --> q2_1_3_c0 <EndOfData Data>
q1_1_3_c0 <Complete> --> q2_1_3_c0 <EndOfData Complete>
q1_1_3_c0 <Ack> --> q2_1_3_c0 <EndOfData Ack>
q1_1_3_c0 <z> --> q2_1_3_c0 <EndOfData z>
q1_1_3_c0 <RawData> --> q1_2_3_c0 <Data RawData>
q1_1_3_c0 <EndOfData> --> q1_2_3_c0 <Data EndOfData>
.....
q4_5_3_c0 <RawData> --> q4_5_4_c0 <Ack RawData>
q4_5_3_c0 <EndOfData> --> q4_5_4_c0 <Ack EndOfData>
q4_5_3_c0 <Start> --> q4_5_4_c0 <Ack Start>
q4_5_3_c0 <Data> --> q4_5_4_c0 <Ack Data>
q4_5_3_c0 <Complete> --> q4_5_4_c0 <Ack Complete>
q4_5_3_c0 <Ack> --> q4_5_4_c0 <Ack Ack>
q4_5_3_c0 <z> --> q4_5_4_c0 <Ack z>
#self-transition from final state(s)
q4_5_4_c0 <z> --> q4_5_4_c0 <z>
```

Figure 7.2: FMUS service: part of pushdown system model for MOPED

```
      ( <> ( q4_5_4_c0 && z ) ) && ( <> push_RawData ) &&
      ( <> pop_RawData ) && ( <> push_Data ) && ( <> pop_Data )
```

Figure 7.3: FMUS service: LTL formula for pushdown model checking

```
--- START ---
q0_0_0_c0 <z>
q1_0_0_c0 <RawData z>
q1_1_0_c0 <z>
q1_2_0_c0 <Data z>
q1_2_0_c0 <push_RawData Data z>
q1_2_0_c0 <RawData Data z>
q1_3_0_c0 <pop_RawData Data z>
q1_3_0_c0 <Data z>
q1_2_0_c0 <push_Data Data z>
q1_2_0_c0 <Data Data z>
q1_2_0_c0 <push_RawData Data Data z>
q1_2_0_c0 <RawData Data Data z>
q1_3_0_c0 <pop_RawData Data Data z>
q1_3_0_c0 <Data Data z>
q2_3_0_c0 <EndOfData Data Data z>
q3_3_0_c0 <Start EndOfData Data Data z>
q3_3_1_c0 <EndOfData Data Data z>
q3_2_1_c0 <push_Data EndOfData Data Data z>
q3_2_1_c0 <Data EndOfData Data Data z>
q3_2_2_c0 <EndOfData Data Data z>
q3_4_2_c0 <Data Data z>
q3_4_2_c0 <pop_Data Data z>
q3_4_2_c0 <Data z>
q3_4_2_c0 <pop_Data z>
q3_4_2_c0 <z>
q3_5_2_c0 <Complete z>
q3_5_3_c0 <z>
q3_5_4_c0 <Ack z>
q4_5_4_c0 <z>
q4_5_4_c0 <z>

--- LOOP ---
q4_5_4_c0 <z>
```

Figure 7.4: FMUS service: counterexample

```
---adaptor
------num_state : 15
------num_T_i   : 9
------num_T_o   : 8
------num_T_h   : 0
---------T_i[0] : ( 0 < 0 > --> 1 < 1, 0 > )
---------T_i[1] : ( 2 < 0 > --> 3 < 2, 0 > )
---------T_i[2] : ( 3 < 2 > --> 3 < 1, 2 > )
---------T_i[3] : ( 4 < 2 > --> 3 < 2, 2 > )
---------T_i[4] : ( 4 < 2 > --> 5 < 3, 2 > )
---------T_i[5] : ( 5 < 3 > --> 6 < 4, 3 > )
---------T_i[6] : ( 7 < 3 > --> 8 < 2, 3 > )
---------T_i[7] : ( 10 < 0 > --> 11 < 5, 0 > )
---------T_i[8] : ( 12 < 0 > --> 13 < 6, 0 > )
---------T_o[0]  : ( 1 < 1 > --> 2 < -1, -1 > )
---------T_o[1]  : ( 3 < 1 > --> 4 < -1, -1 > )
---------T_o[2]  : ( 6 < 4 > --> 7 < -1, -1 > )
---------T_o[3]  : ( 8 < 2 > --> 9 < -1, -1 > )
---------T_o[4]  : ( 9 < 3 > --> 10 < -1, -1 > )
---------T_o[5]  : ( 10 < 2 > --> 10 < -1, -1 > )
---------T_o[6]  : ( 11 < 5 > --> 12 < -1, -1 > )
---------T_o[7]  : ( 13 < 6 > --> 14 < -1, -1 > )
```

Figure 7.5: FMUS service: adaptor (tool output)

$$D = (Q_D, q_D^0, \Gamma, z, T_D, F_D)$$

where

- $Q_D = \{\ q0_D,\ q1_D,\ q2_D,\ q3_D,\ q4_D,\ q5_D,\ q6_D,\ q7_D,\ q8_D,\ q9_D,$
  $q10_D,\ q11_D,\ q12_D,\ q13_D,\ q14_D\ \}.$
- $q_D^0 = q0_D.$
- $\Gamma = \{\ z,\ RawData,\ Data,\ EndOfData,\ Start,\ Complete,\ Ack\ \}.$
- $z \in \Gamma$ is initial symbol of stack.
- $T_D = \{\ (q0_D,\ z) \hookrightarrow (q1_D,\ \ < RawData\ z >),$
  $(q2_D,\ z) \hookrightarrow (q3_D,\ \ < Data\ z >),$
  $(q3_D,\ Data) \hookrightarrow (q3_D,\ \ < RawData\ Data >),$
  $(q4_D,\ Data) \hookrightarrow (q3_D,\ \ < Data\ Data >),$
  $(q4_D,\ Data) \hookrightarrow (q5_D,\ \ < EndOfData\ Data >),$
  $(q5_D,\ EndOfData) \hookrightarrow (q6_D,\ \ < Start\ EndOfData >),$
  $(q7_D,\ EndOfData) \hookrightarrow (q8_D,\ \ < Data\ EndOfata >),$
  $(q10_D,\ z) \hookrightarrow (q11_D,\ \ < Complete\ z >),$
  $(q12_D,\ z) \hookrightarrow (q13_D,\ \ < Ack\ z >),$
  $(q1_D,\ RawData) \hookrightarrow (q2_D,\ \epsilon),$
  $(q3_D,\ RawData) \hookrightarrow (q4_D,\ \epsilon),$
  $(q6_D,\ Start) \hookrightarrow (q7_D,\ \epsilon),$
  $(q8_D,\ Data) \hookrightarrow (q9_D,\ \epsilon),$
  $(q9_D,\ EndOfData) \hookrightarrow (q10_D,\ \epsilon),$
  $(q10_D,\ Data) \hookrightarrow (q10_D,\ \epsilon),$
  $(q11_D,\ Complete) \hookrightarrow (q12_D,\ \epsilon),$
  $(q13_D,\ Ack) \hookrightarrow (q14_D,\ \epsilon)\ \}$
- $F_D = Q_D.$

Figure 7.6: FMUS service: adaptor as an IPS

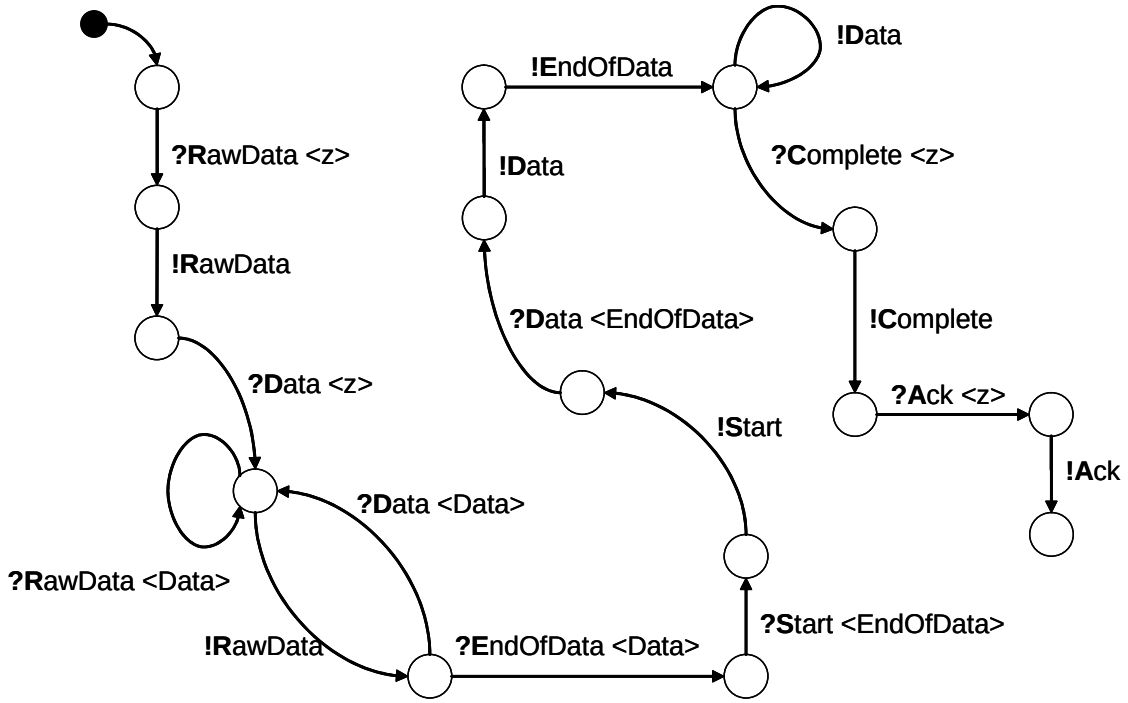


Figure 7.7: FMUS service: adaptor

confirm if the modeling of adaptors is applicable on implementing real application. In this section, the generated adaptor of the FMUS service is used in implementation and we choose BPEL as the development platform of the adaptor of the FMUS service. The reasons of choosing BPEL are as follows:

- Technologies of web services are more popular, i.e., state of the art. than traditional software components. Furthermore, web services are considered next generation of software components.
- Definition languages of protocol of web services are open standards. Generally, there is no problem in representing protocols of web services by IA4ADs introduced in the approach. Therefore, if the implementation of adaptors modeled by pushdown systems in the approach is confirmed feasible on developing platforms of web services, everyone can perform adaptor generation for web services by following the approach.

The idea of implementing an adaptor in BPEL platform is quite simple. Since an adaptor is represented by an IPS which is generally a pushdown system, we may treat a pushdown system an finite state machine with a stack interacting with it. Therefore, it should work by implementing two processes: one for the finite state machine part of the pushdown system, and one for the stack part of the pushdown system. For the FMUS service, the structure of the adapted system is then as shown in Fig. 7.8.

Since the approach only provides adaptor generation for given behavior interfaces of components and the behavior interfaces are quite abstracted to meet level of details of

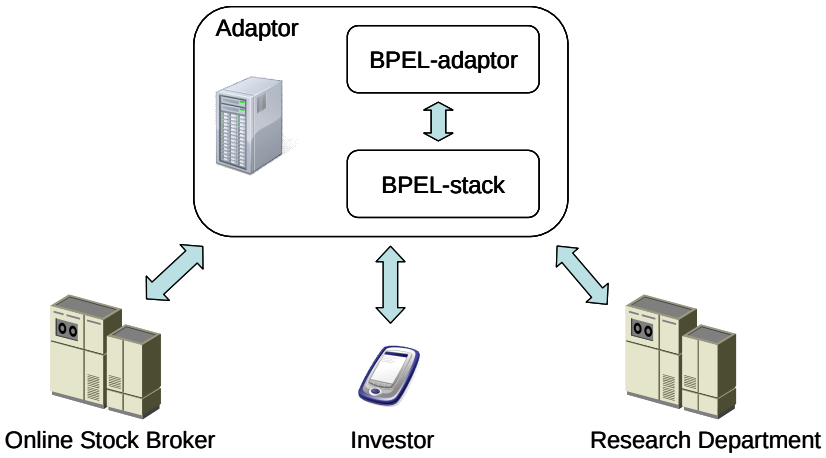


Figure 7.8: FMUS service: BPEL adaptor

implementation of BPEL process. We only provide directions of how to implement generated adaptors as BPEL services. The very details such type of messages or variables in messages should depend on problems of real applications that take the approach of adaptor generation. Now we give directions of implementing BPEL processes for generated adaptors. Basically, an adaptor is implemented as two BPEL process adaptor and built two BPEL processes: *BPEL-adaptor* represents the finite state machine part of the adaptor and *BPEL-stack* represents the stack part of the adaptor. Directions of implementing the two BPEL processes are as follows:

- **Building BPEL-adaptor**

  This BPEL process captures the finite state machine part of adaptor. Note that BPEL is a description language for workflow so that all states are implemented as activities while transitions are implemented as service actions (invoke/receive). The activity after a receive action should push the received message into the stack by call the push action of the BPEL-stack process. The activity before an invoke action should prepare the message to be sent by calling the pop action of the BPEL-stack process. When building a BPEL process, all messages require corresponding port types for sending and receiving defined in WSDL [18]. Messages can be implemented as data types such as integer, string, or composite data type depending on signatures of given services.

- **Building BPEL-stack**

  This BPEL process performs functionalities of a standard stack. The stack content can be defined as an array of user-defined message type and its property *MaxOccurs* has to be set to unbounded. Thus, the push action receives a message and add it into the array with increasing the length of array by one in the next activity. The implementation of pop transition is similar but length of array is decreased by one.

By following the above directions, we implemented the generated adaptor shown in Fig. 7.7 as two BPEL processes. The finite state machine part is partly shown in Fig. 7.9

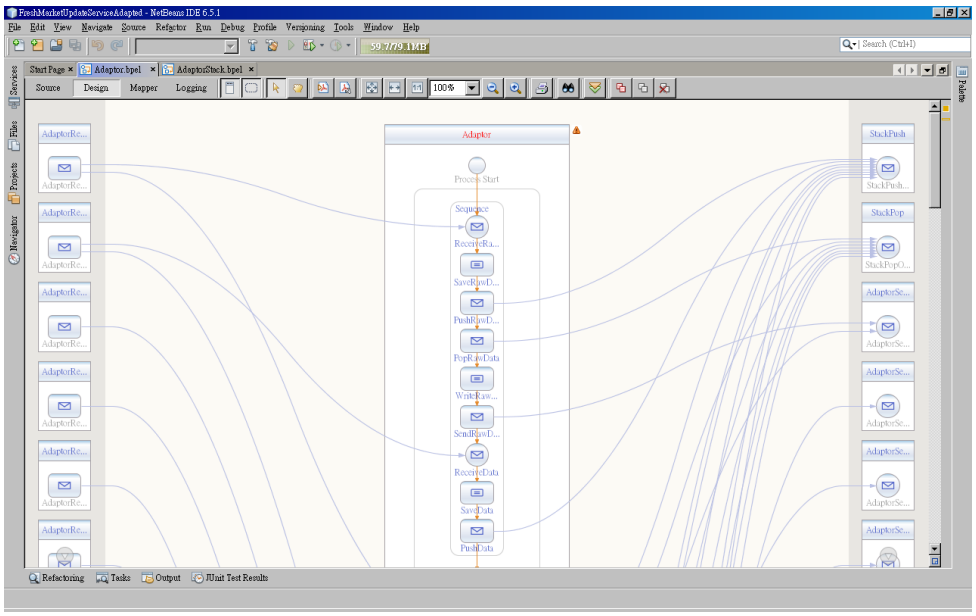


Figure 7.9: FMUS service: BPEL-adaptor process

and the stack part is partly shown in Fig. 7.10. The implementation is down using Netbeans IDE [19] which provides graphical editing and testing environment for developing BPEL process. For testing the adapted system of the FMUS service, we also implemented the three services *Online Stock Broker*, *Research Department*, and *Investor* as BPEL processes and assembled a composite application shown in Fig. 7.11. Some test cases are specified and the tests were successful. Therefore, we may conclude that the approach is applicable on web services in BPEL platform. For given components whose protocols are specified as BPEL processes, the approach can support the adaptor generation for these components if needed. The generated adaptor can be then implemented as BPEL processes and composed with the components to build an adapted system.

7.3 General Cases of Adaptation

The FMUS service is a basic ordering behavioral mismatching problem with unbounded messages. Note that the FMUS service is also the motivational example of this work. It is obvious that the approach can solve this problem and we gave some evidences to proof this in the experiment with the FMUS service in section 7.1. However, since we are dealing with adaptation of software component/web services, general cases of adaptation problems should also be examined and see if the approach is useful or not in these cases. By the survey of adaptation problems, we conclude that two general cases of adaptation problems should be considered:

- There are *signature mismatches* in behavior interfaces of components.
- There are *branchings* in behavior interfaces of components.

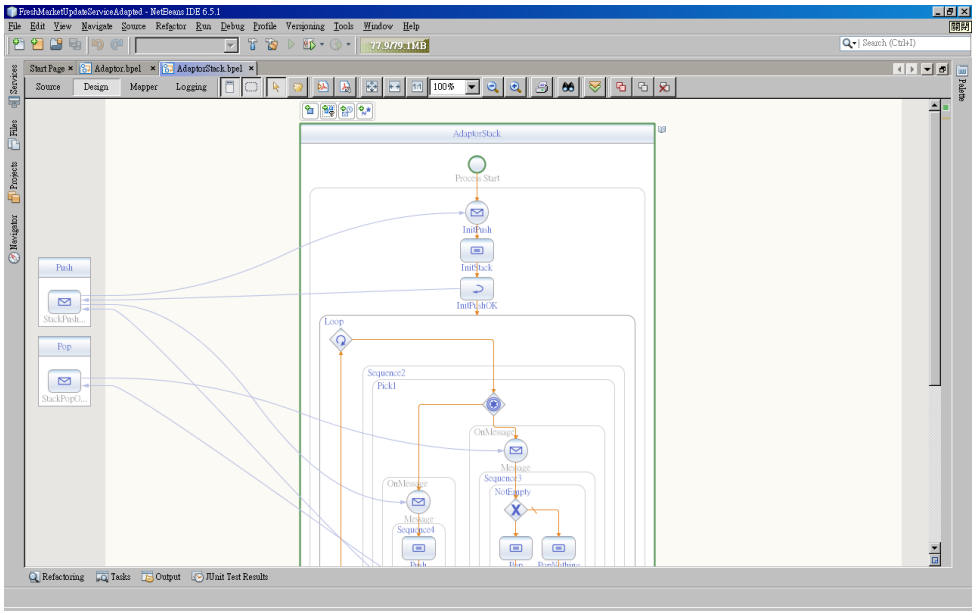


Figure 7.10: FMUS service: BPEL-stack process

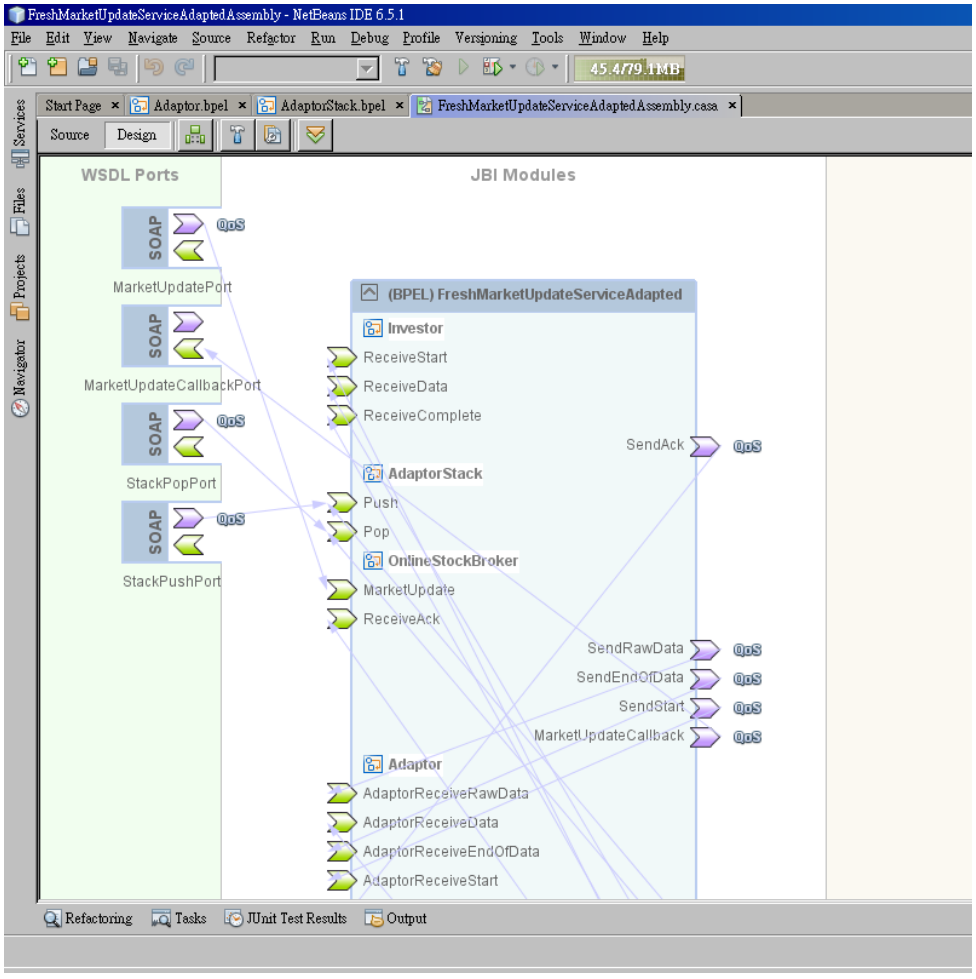


Figure 7.11: FMUS service: assemble and test

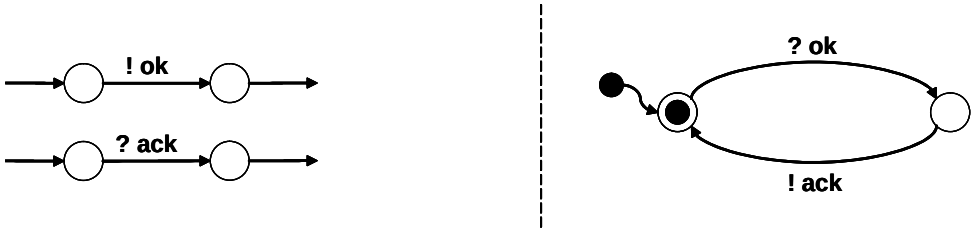
It is obvious that the above two cases can show separately or together. In the following sections, the two general cases will be discussed with an example given for demonstration of applying the approach of adaptation on the two general cases.

7.3.1 Signature Mismatches

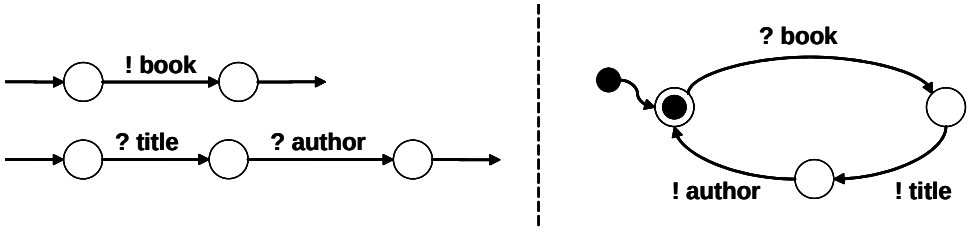
In the approach, we assume that there is no signature mismatch in given components for adaptor generation or signature mismatches are solved somehow before applying the approach. However, one should admit that signature mismatches are very common in adaptation problems. In conventional approach of adaptation, developers can specify different mappings of labels in different adaptation contracts to generate different adaptors for different cases of the same system.

Though the approach is not designed to deal with generating mapping of labels for signature mismatches, it is necessary for the approach to give strategies about how to apply the approach on components with signature mismatches. Basically, when given a group of components with signature mismatches, names of some messages are not match so that the components can not pass the compatibility check and therefore the approach is not applicable. Therefore, what we need to do is to make the group of components pass the compatibility check. The idea of doing this is introducing special components that represent specified mappings of labels. Note that the mapping of labels are specified by developers so that correctness of the mappings is out of the scope of the approach. If the mapping of labels are correctly specified, the given components along with the special components representing these mappings should pass the compatibility check. Then we can apply the approach and proceed to adaptor generation.

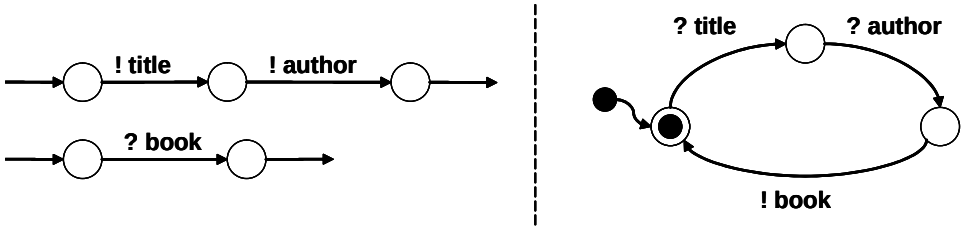
We may call the components representing specified mapping of labels *Mapping Components*. Since mappings of labels can vary depending on problems, we only provide general cases of mappings and the corresponding mapping components. Here three cases of mappings are considered and the demonstrations of building corresponding mapping components are shown in Fig. 7.12. In each case, the left part shows behavior segments of two components while the right part shows a mapping component that provides signature mapping for the two behavior segments. The first case shown in Fig. 7.12(a) is a mapping of different message names. In this case, `ok` sent from one service is expected to synchronize with `ack` received by another service. A mapping component is designed to receive `ok` and delivers `ack` so that the synchronization can be done through this mapping component. The second case shown in Fig. 7.12(b) is a message splitting. Information of a book in message `book` is sent by a message while another component receives the information of the book in two messages `title` and `author`. Thus, a mapping component representing this message split should receive the message `book` and separate the information in `book` into two messages `title` and `author`, then deliver the two messages. The third case shown in Fig. 7.12(c) is a merging of messages. This case uses the same message names in second case but opposite direction in communication. In this case, one component sends information of a book using two messages of `title` and `author` while another component receives the information in one message `book`. Therefore, a mapping component should be designed to receive `title` and `author` sequentially to merge the two message and create `book` for delivery. It should be easy to understand that all these three cases are not difficult for developers to figure out the mismatches and design corresponding mapping components as long as appropriate specifications of mappings of labels are



(a) message mapping



(b) message splitting



(c) message merging

Figure 7.12: Examples of mapping components

given. Furthermore, the difficulty of designing mapping service does not increase along with the scale of system of components since we only need to statically check all messages one by one without worrying how to synchronize them. However, it should be warned that in this work, there is no standard for the design of mapping services. There may be other cases of signature mismatches as well mapping components for them though Fig. 7.12 already demonstrates most frequently appeared cases. Also, it should be noted that mapping components may effect the result of adaptation generation. This means, adaptor generation by the approach may be fail because improperly specified mapping components.

To show the case of components with signature mismatches, Fig 7.13 shows an example of two components with signature mismatches. This example is a motivational example in work of J. Martín and E. Pimentel [20] which aims on automatic generation of adaptation contracts by heuristic searching algorithms. We may call the problem the *File Download service* or the *FD service* in short. The FD service has two components: the *Client* component sends information of user name `UserName` and password `Password` to login to the server. the *Client* then sends request of file downloading `Download` to the server and receive the content of file in message `Data`. On the other hand, the *Server* component first takes login from the client in message `Login`. Then the *Service* component may receive

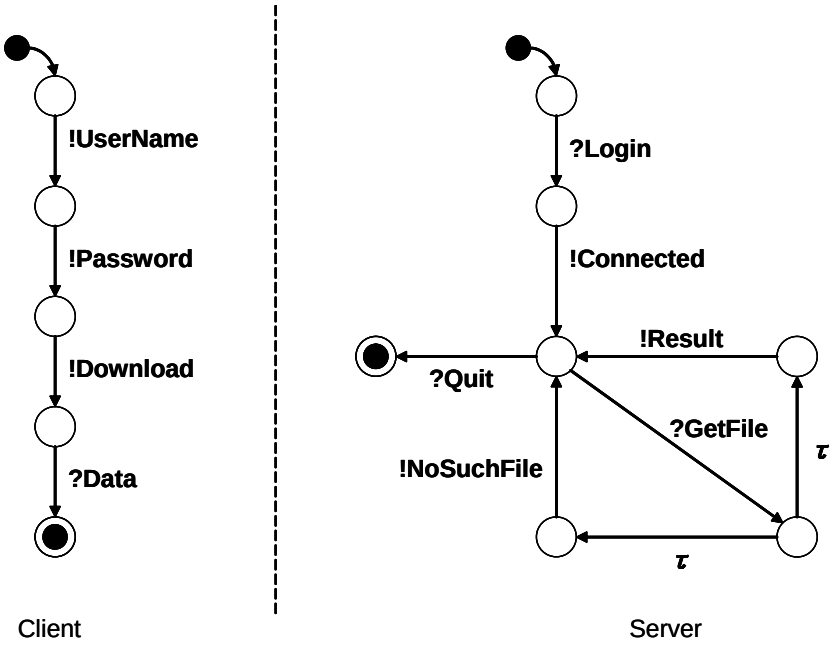


Figure 7.13: File download service

aborting message `Quit` to end at this point or receive `GetFile` request to start providing file downloading procedure. Then the *Server* may return `Result` which is the content of the file or an error message `NoSuchFile` to inform that the download fails.

The two components in the FD service have quite many signature mismatches. In order to make the two components cooperate together seamlessly, we prepare six mappings of labels. These mappings are represented as six mapping components shown in Fig. 7.14. *Mapping1* is a message merging that merges `UserName` and `Password` from the *Client* into `Login` to the *Server*. *Mapping2* is just a absorber of message `Connected` since client is not designed to take this message. *Mapping3* is a mapping for message `Download` from the *Client* to message `GetFile` for the *Server* to match the difference of names of requests. *Mapping4* is a mapping that matches the file content `Result` from the *Service* with `Data` to the *Client*. *Mapping5* is a mapping that matches the error message `NoSuchFile` from the *Service* with `Data` for the *Client*. *Mapping6* is a message generator which generates message `Quit` since the *Client* does not designed to send this message. The two components and the six mapping components can be encoded as input of the approach shown in Fig. 7.15.

By introducing the six mapping components, the system of FD service can now pass the compatibility check of the approach. Note that actually *Mapping4* and *Mapping5* violate the compatibility condition because they both has `Data` in the set of output alphabets. Also, all mapping components do not follow the constraint of an IA4AD that the initial and the final states should be different states. However, this is allowed in the approach since mapping components are special components just introduced for the purpose of representing mappings of labels. Generally, a mapping might be applied more than one time or even not applied at all. This is not a problem since we do not care about

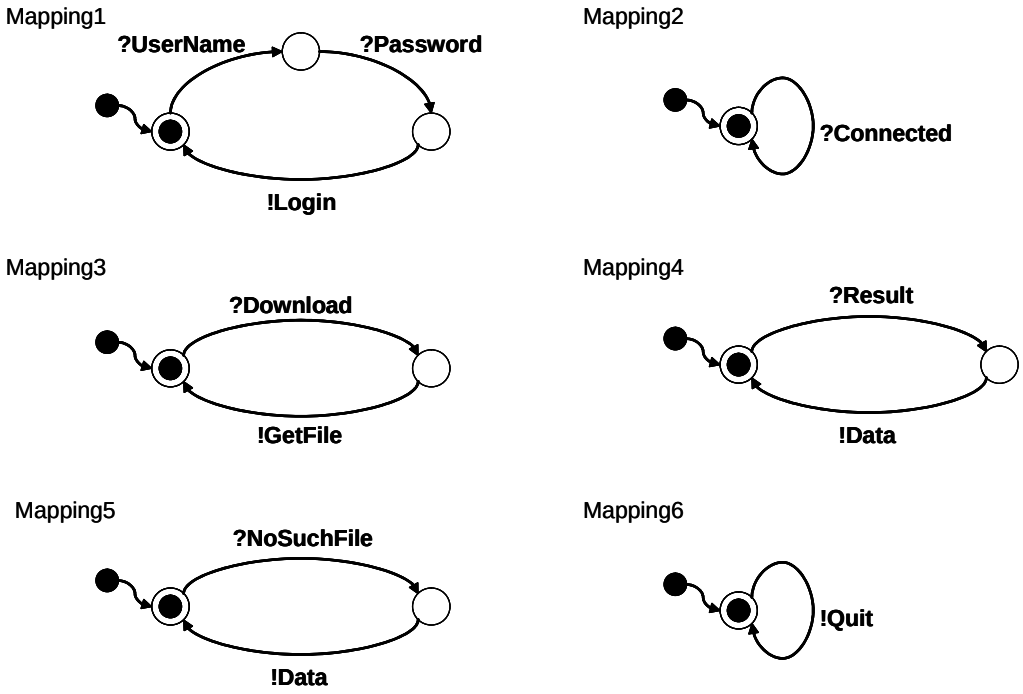


Figure 7.14: File download service: mapping components

whether a mapping component finish its functionalities as long as given components do execute and achieve their functionalities. Thus, looped structure that initial and final state are the same state is necessary for building mapping components. Also, warnings from compatibility check on mapping components are neglected.

Now we may proceed to the part of detection of behavior mismatches using SPIN. The tool of the approach reads the input file and generateds a Promela model for SPIN. In the Promela model, two propositions `accept_Client` and `accept_Server` are defined and used in the LTL formula representing the property of Behavior Mismatch Free:

```
( <> accept_Client ) && ( <> accept_Server )
```

The result of model checking by SPIN for the property of Behavior Mismatch Free returns a counterexample which is a trail shown in Fig. 7.16. By examining the trail, one may understand that though all mapping components seem correctly specified, behavior mismatches still exist in the case that the *Server* receives `Quit` and ends the file download service while the *Client* does not receive any data and still waits for file content. Since *Mapping6*, message `Quit` can be sent anytime, it is possible that the *Server* ends its service while the *Client* does not receive anything. Thus, an adaptor that make sure the *Client* executes to its final state is needed. At this point, we may conclude that introducing mapping components do solve signature mismatches so that the approach is applicable in systems with signature mismatches.

We may now proceed to adaptor generation of the approach using pushdown model checking by MOPED. Unfortunately, the adaptor generation failed because of another problem: *branchings* in the behavior interface of the *Server*. We will discuss this issue in Section 7.3.2.

```
# File Download Service

service:: Client
init::S0
final::S4
(S0,!UserName,S1) (S1,!Password,S2) (S2,!Download,S3) (S3,?Data,S4)

service:: Server
init::S0
final::S6
(S0,?Login,S1) (S1,!Connected,S2) (S2,?GetFile,S3) (S3,tau,S4)
(S3,tau,S5) (S4,!Result,S2) (S5,!NoSuchFile,S2) (S2,?Quit,S6)

service::Mapping1
init::S0
final::S0
(S0,?UserName,S1) (S1,?Password,S2) (S2,!Login,S0)

service::Mapping2
init::S0
final::S0
(S0,?Connected,S0)

service::Mapping3
init::S0
final::S0
(S0,?Download,S1) (S1,!GetFile,S0)

service::Mapping4
init::S0
final::S0
(S0,?Result,S1) (S1,!Data,S0)

service::Mapping5
init::S0
final::S0
(S0,?NoSuchFile,S1) (S1,!Data,S0)

service::Mapping6
init::S0
final::S0
(S0,!Quit,S0)
```

Figure 7.15: File download service: input file

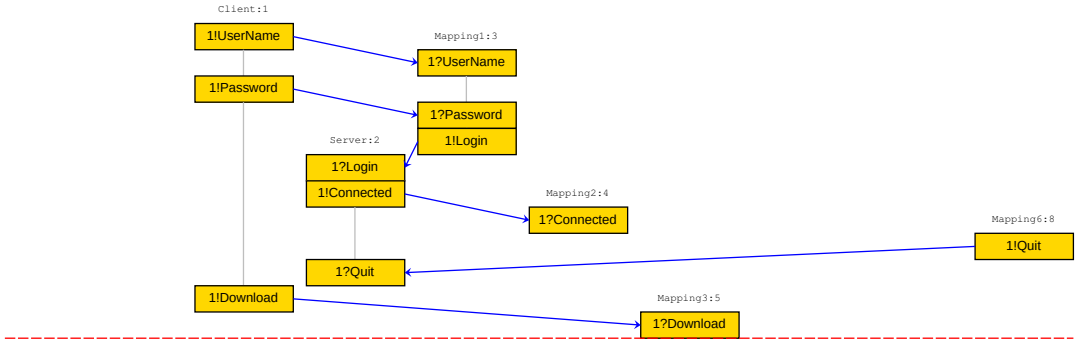


Figure 7.16: File download service: SPIN output

7.3.2 Branchings

Branchings are different route of execution in the behavior interface of a component. Basically, a component is designated to execute a sequence of tasks for achieving some functionalities. However, a component more often is designed to execute only part of its tasks based on interactions with other components. For example, a component providing several functionalities may not perform all the functionalities in one execution but instead perform some of the functionalities based on requests from other components. Therefore, a component usually does not have only one designed execution trace but multiple and exclusive branchings of execution traces. In the FD service shown in Fig. 7.13, the behavior interface of the *Server* component has branchings in two of its states: the state which has two outgoing input transitions receiving `Quit` and `GetFile` separately, and the state which has two outgoing internal transitions leading to output transitions sending `Quit` and `GetFile` separately. The two states are state `S2` and state `S3` in the input file shown in Fig. 7.15.

Unfortunately, since adaptor generation in the approach uses a counterexample returned from pushdown model checking to build the adaptor for given components, branchings in behavior interfaces of components may cause problems. Recall that in Section 3.2, components are represented by IA4ADs where each IA4AD has only one initial state and one final state and Coordinator is built as an one state IPS, the adapted synchronous composition of components with Coordinator is then an IPS having also one initial state and one final state. We may imagine that the adapted synchronous composition with Coordinator is as shown in Fig. 7.17. Note that there are only one initial and on final state in this system behavior. Also, it should be noted that in Fig. 7.17, two branchings started from a state go through two push transitions `?a <c>` and `?b <>` while other transitions are abstracted by dash lines. Therefore, one may easily understand that a counterexample can only go through one branching, that is, either through transition `?a <c>` or through transition `?b <>`. However, we usually need an adaptor that can cover all branchings so that all functionalities can be achieved. For example, in the system of FD service, we need all branchings executable in the generated adaptor so that the *Server* component can perform all designated behavior. If one branching, for example, the branching goes through output transition sending `NoSuchFile` is missing in the generated adaptor, then the adaptor generation is considered failed since this adaptor can not handle the situation when file download fails in server. Furthermore, in the FD service,

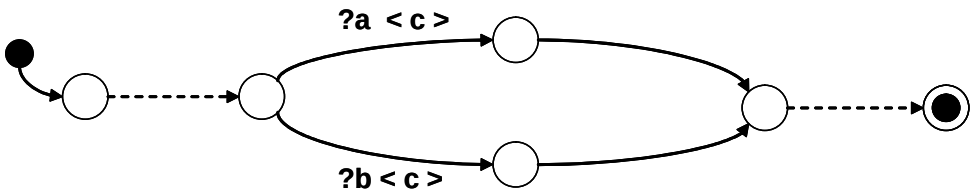


Figure 7.17: Branchings: the problem

messages `GetFile`, τ, `Result`, and `NoSuchFile` are collected as *Unbounded Messages* and corresponding *Looped Transitions* are specified as fairness property. Since transitions relating to exclusive branchings are included in Looped Transitions transitions, we get no counterexample since no trace can go through both exclusive branchings being specified as fairness property.

To solve the problem caused by branchings, we need some modification to the adaptor generation process of the approach. If we want the pushdown model checking searching through every exclusive branches, the system behavior has to be modified so that searching algorithms of pushdown model checking can start over from the initial state again when it searched one branching and reaches the final state. This can be achieved by simply adding an epsilon transition $(q^f, z) \hookrightarrow (q_0, z)$ to establish a connection from the final state to the initial state, which is demonstrated in Fig. 7.18. Note that this epsilon transition is executable only when the final state is reached with the condition of empty stack. This guarantees that the property of Behavior Mismatch Free is always satisfied each time the system behavior is searched by the pushdown model checking algorithms for a branching. With this modification of adding a epsilon transition, we may add fairness properties for corresponding transitions of all branchings to the property ϕ necessary for adaptors. Thus, the pushdown model checking algorithms can search the system behavior with exclusive branchings and generate a counterexample which travels through multiple executions of the system behavior to guarantee all exclusive branchings are visited. Then we can build an adaptor which supports exclusive branchings form this counterexample. Transitions corresponding to branchings are called *Branching Transitions*. Branching Transitions are easy to be collected. We only need to look for states with two or more outgoing transitions and these outgoing transitions are then *Branching Transitions*. Fairness of Branching Transitions can be specified using the same technique of specifying fairness property of Looped Transitions for Unbounded Messages.

By the modification of adding an epsilon transition, we can now deal with branchings in the FD service. The epsilon transition added in the system behavior as last transition rule is shown below:

```
#epsilon-transition from final state(s) to initial state
q4_6_0_0_0_0_0_0_c0 <z> --> q0_0_0_0_0_0_0_0_c0 <z>
```

Thus, the pushdown system model representing the system behavior is now modified to let the pushdown model checking algorithm search through all exclusive branchings. The LTL formula representing the property ϕ, which consists of the property of Behavior

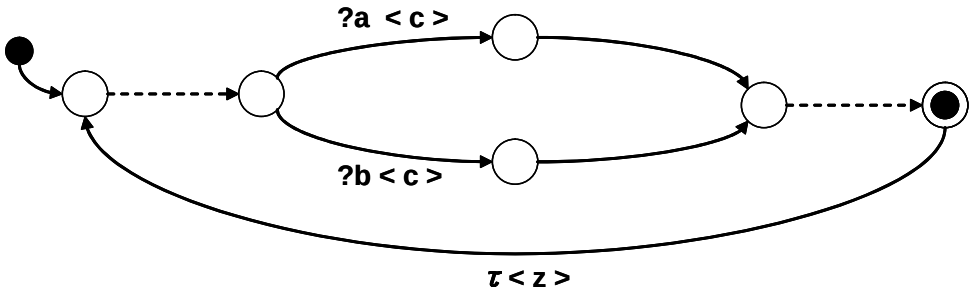


Figure 7.18: Branchings: the idea

Mismatch Free, the property of Unbounded Messages, i.e., Looped Transitions, and the property of Branching Transitions is listed below:

```
( <> ( q4_6_0_0_0_0_0_0_c0 && z ) ) &&
( <> pop_GetFile ) && ( <> pop_Quit ) &&
( <> push_Result ) && ( <> push_NoSuchFile ) &&
( <> tau_1_0 ) && ( <> tau_1_1 )
```

One may recognize that in this formula, the first line shows the property of Behavior Mismatch Free where `q4_6_0_0_0_0_0_0_c0` is the final state. The rest are fairness property of Looped and Branching Transitions. Note that the last two fairness property for `tau_1_0` and `tau_1_1` are the internal transitions in the *Server* component and are distinguished by adding different indexes since these two transitions are not the same transition while sharing the same label. Then MOPED is applied for pushdown model checking the modified system behavior for the above property and a counterexample is returned as follows:

```
--- START ---
q0_0_0_0_0_0_0_c0 <z>
q1_0_0_0_0_0_0_c0 <UserName z>
q1_0_1_0_0_0_0_c0 <z>
q2_0_1_0_0_0_0_c0 <Password z>
q2_0_2_0_0_0_0_c0 <z>
q3_0_2_0_0_0_0_c0 <Download z>
q3_0_2_0_1_0_0_c0 <z>
q3_0_0_0_1_0_0_c0 <Login z>
q3_1_0_0_1_0_0_c0 <z>
q3_2_0_0_1_0_0_c0 <Connected z>
q3_2_0_0_0_0_0_c0 <GetFile Connected z>
q3_3_0_0_0_0_0_c0 <pop_GetFile Connected z>
q3_3_0_0_0_0_0_c0 <Connected z>
q3_4_0_0_0_0_0_c0 <tau_1_0 Connected z>
q3_4_0_0_0_0_0_c0 <Connected z>
q3_2_0_0_0_0_0_c0 <push_Result Connected z>
q3_2_0_0_0_0_0_c0 <Result Connected z>
```

```
q3_2_0_0_0_1_0_0_c0 <Connected z>
q3_2_0_0_0_0_0_0_c0 <Data Connected z>
q4_2_0_0_0_0_0_0_c0 <Connected z>
q4_2_0_0_0_0_0_0_c0 <z>
q4_2_0_0_0_0_0_0_c0 <Quit z>
q4_6_0_0_0_0_0_0_c0 <pop_Quit z>
q4_6_0_0_0_0_0_0_c0 <z>
q0_0_0_0_0_0_0_0_c0 <z>
q1_0_0_0_0_0_0_0_c0 <UserName z>
q1_0_1_0_0_0_0_0_c0 <z>
q2_0_1_0_0_0_0_0_c0 <Password z>
q2_0_2_0_0_0_0_0_c0 <z>
q3_0_2_0_0_0_0_0_c0 <Download z>
q3_0_2_0_1_0_0_0_c0 <z>
q3_0_0_0_1_0_0_0_c0 <Login z>
q3_1_0_0_1_0_0_0_c0 <z>
q3_2_0_0_1_0_0_0_c0 <Connected z>
q3_2_0_0_0_0_0_0_c0 <GetFile Connected z>
q3_3_0_0_0_0_0_0_c0 <pop_GetFile Connected z>
q3_3_0_0_0_0_0_0_c0 <Connected z>
q3_5_0_0_0_0_0_0_c0 <tau_1_1 Connected z>
q3_5_0_0_0_0_0_0_c0 <Connected z>
q3_2_0_0_0_0_0_0_c0 <push_NoSuchFile Connected z>
q3_2_0_0_0_0_0_0_c0 <NoSuchFile Connected z>
q3_2_0_0_0_0_1_0_c0 <Connected z>
q3_2_0_0_0_0_0_0_c0 <Data Connected z>
q4_2_0_0_0_0_0_0_c0 <Connected z>
q4_2_0_0_0_0_0_0_c0 <z>
q4_2_0_0_0_0_0_0_c0 <Quit z>
q4_6_0_0_0_0_0_0_c0 <pop_Quit z>
q4_6_0_0_0_0_0_0_c0 <z>
q0_0_0_0_0_0_0_0_c0 <z>

--- LOOP ---
q1_0_0_0_0_0_0_0_c0 <UserName z>
q1_0_1_0_0_0_0_0_c0 <z>
q2_0_1_0_0_0_0_0_c0 <Password z>
q2_0_2_0_0_0_0_0_c0 <z>
q2_0_0_0_0_0_0_0_c0 <Login z>
q2_1_0_0_0_0_0_0_c0 <z>
q2_2_0_0_0_0_0_0_c0 <Connected z>
q3_2_0_0_0_0_0_0_c0 <Download Connected z>
q3_2_0_0_1_0_0_0_c0 <Connected z>
q3_2_0_0_0_0_0_0_c0 <GetFile Connected z>
q3_3_0_0_0_0_0_0_c0 <pop_GetFile Connected z>
q3_3_0_0_0_0_0_0_c0 <Connected z>
q3_5_0_0_0_0_0_0_c0 <tau_1_1 Connected z>
```

```
q3_5_0_0_0_0_0_0_c0 <Connected z>
q3_2_0_0_0_0_0_0_c0 <push_NoSuchFile Connected z>
q3_2_0_0_0_0_0_0_c0 <NoSuchFile Connected z>
q3_2_0_0_0_0_1_0_c0 <Connected z>
q3_2_0_0_0_0_1_0_c0 <z>
q3_2_0_0_0_0_0_0_c0 <Data z>
q3_2_0_0_0_0_0_0_c0 <Quit Data z>
q3_6_0_0_0_0_0_0_c0 <pop_Quit Data z>
q3_6_0_0_0_0_0_0_c0 <Data z>
q4_6_0_0_0_0_0_0_c0 <z>
q0_0_0_0_0_0_0_0_c0 <z>
```

It is easy to understand that after the modification for support branchings, the returned counterexample is a long trace since now this trace is obtained from searching on multiple executions of the system behavior. One may notice that in the counterexample, segment of two configurations can be found as follows:

```
q4_6_0_0_0_0_0_0_c0 <z>
q0_0_0_0_0_0_0_0_c0 <z>
```

The two configuration shows that the trace does travel from the final state to the initial state through the epsilon transition added for supporting branchings. Finally, we may build an adaptor from the counterexample and draw in graph using Graphviz as shown in Fig. 7.19. With careful examination of the behavior of the adaptor, it should be confirmed that the behavior of the adaptor satisfies the property ϕ of the FD service. Therefore, we may conclude that the approach can still support adaptor generation for components with signature mismatches and branchings in their behavior interfaces.

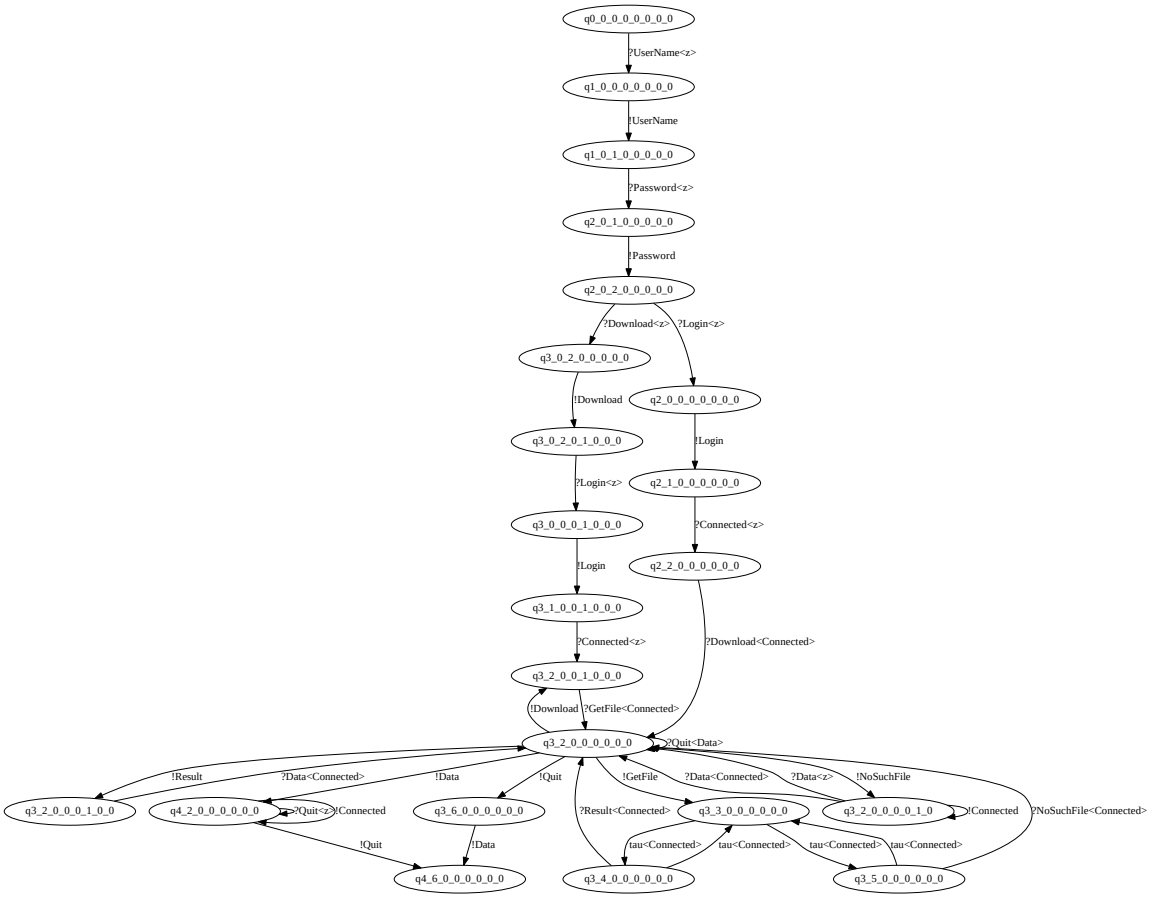


Figure 7.19: FD service: adaptor drawn by Graphviz

Chapter 8

Evaluation

For summarizing/concluding the experiments in Chapter 7 and evaluating the approach, we would like to demonstrate one more adaptation problem which has all issues discussed in Section 7.3: behavior mismatches with non-regular interactions, signature mismatches, and branchings in behavior interfaces of components. We may call an adaptation problem consists of all the three issues a *fully mismatching* problem and try applying the approach to solve it. However, since adaptation problems which require non-regular adaptors are first considered in this work while not being considered in conventional approach of adaptation, fully mismatching adaptation problems are rare. It is difficult to discover from papers of related work an adaptation problem which is fully mismatching. Thus, we decided to make one by ourselves and extend the FMUS service problem by adding signature mismatches as well as branchings in the behavior interfaces of the three services. Then we can apply the approach to solve this *fully mismatching* adaptation problem to see if the approach can solve all the issues together.

The extended FMUS service is shown in Fig. 8.1. This adaptation problem now has exclusive branchings as well as signature mismatches and needs a non-regular adaptor. In this system, *Investor* may send two exclusive requests, i.e., exclusive branchings, of either asking *Online Stock Broker* to send a list of `RawData` to *Research Department* for analysis, which is same as the FMUS service, or asking *Online Stock Broker* to proceed transactions. In the branching of requesting transactions, *Investor* send the target to be proceeded in message `Trade` following with the desired price in message `Quote` while *Online Stock Boker* receives only one message `Transac`. This is a signature mismatch we need to take care. Another signature mismatch is when *Online Stock Broker* sending the result of transaction in message `Record` to *Investor* while messages `Log` is expected by *Investor*. Note that *Research Department* only do the task of analyzing `RawData` so there is nothing for *Research Department* to do when transaction is requested by *Investor*. But *Research Department* still needs to move to its final state through receiving the message `NoRawData`.

Before applying the approach for adaptor generation, first the two signature mismatches have to be taken care by specifying mappings of labels for building mapping components. Fig. 8.2 shows two mapping components we specified for the two signature mismatches. The mapping component in the left merges information of transaction target `Trade` and preferred price `Quote` from *Investor* into `Transac` to send to *Online Stock Broker*. The mapping component in the right takes transaction result `Record` from *Online Stock Broker* and sends `Log` to *Investor*. It also tells *Research Department* there is nothing

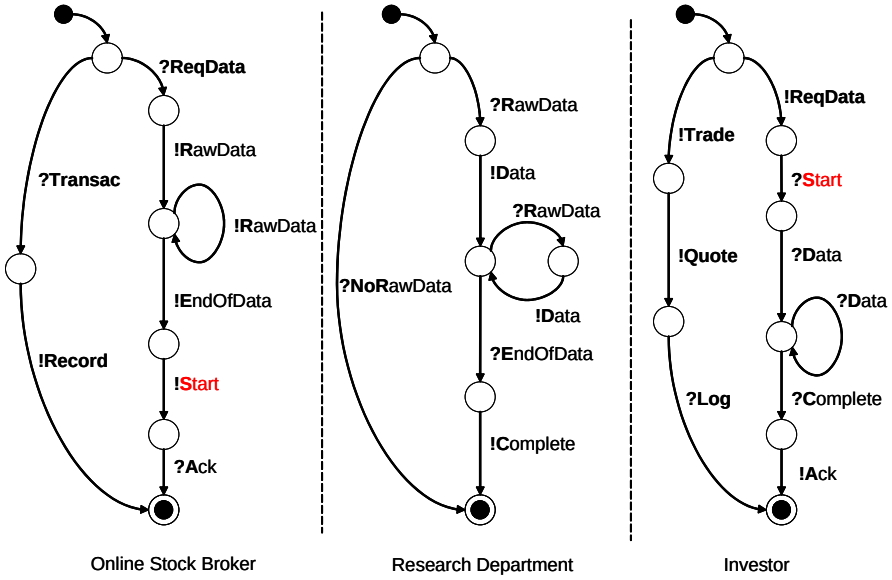


Figure 8.1: Extended FMUS service

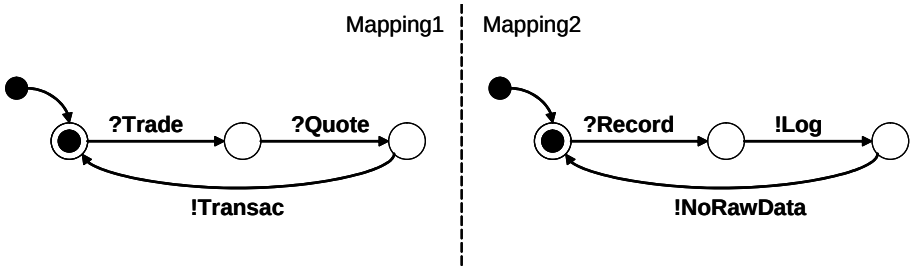


Figure 8.2: Mapping services for message mapping

to analysis by sending `NoRawData` to *Research Department*.

Now we have five components including two mapping components and are ready to apply the approach for adaptor generation. The behavior interfaces of the five components are encoded in an input file shown in Fig. 8.3 for the tool to read and perform computations. First compatibility checks are preformed and the components passed the check. Note that mapping components are not constrained by neither condition of compatibility or condition of IA4ADs. Thus, we may proceed to adaptor generation for solving behavior mismatches.

According behavior interfaces of the five components, the tool generates Coordinator for the system and computes the adapted synchronous composition to build an IPS and output as a pushdown system model for MOPED. Note that here the pushdown system is modified to support branchings as described in Section 7.3.2. Along with the output of the pushdown system model for moped, a LTL formula consists of the property of Behavior Mismatch Free and the property of Looped Transitions and Branching Transitions is also output by the tool. MOPED then checks the system for the property then returns a counterexample. The counterexample is then read by the tool for generating an adaptor.

```
service:: OnlineStockBroker
init::S0
final::S6
(S0,?ReqData,S1) (S1,!RawData,S2) (S2,!RawData,S2)
(S2,!EndOfData,S3) (S3,!Start,S4) (S4,?Ack,S6)
(S0,?Transac,S5) (S5,!Record,S6)

service:: ResearchDepartment
init::S0
final::S5
(S0,?RawData,S1) (S1,!Data,S2) (S2,?RawData,S3)
(S3,!Data,S2) (S2,?EndOfData,S4) (S4,!Complete,S5)
(S0,?NoRawData,S5)

service:: Investor
init::S0
final::S7
(S0,!ReqData,S1) (S1,?Start,S2) (S2,?Data,S3)
(S3,?Data,S3) (S3,?Complete,S4) (S4,!Ack,S7)
(S0,!Trade,S5) (S5,!Quote,S6) (S6,?Log,S7)

service:: Mapping1
init::S0
final::S0
(S0,?Trade,S1) (S1,?Quote,S2) (S2,!Transac,S0)

service:: Mapping2
init::S0
final::S0
(S0,?Record,S1) (S1,!Log,S2) (S2,!NoRawData,S0)
```

Figure 8.3: FMUSv2: the input file

Here we save the space for details of the process of adaptor generation and directly show the final result in Fig. 8.4. We may confirm that the two branches in the extended FMUS service are correctly showed in the behavior of the adaptor. We may also confirm that the non-regular behavior is correctly generated in one branchings. Thus, we may conclude that the approach is able to generate adaptors for *fully mismatching* adaptation problems.

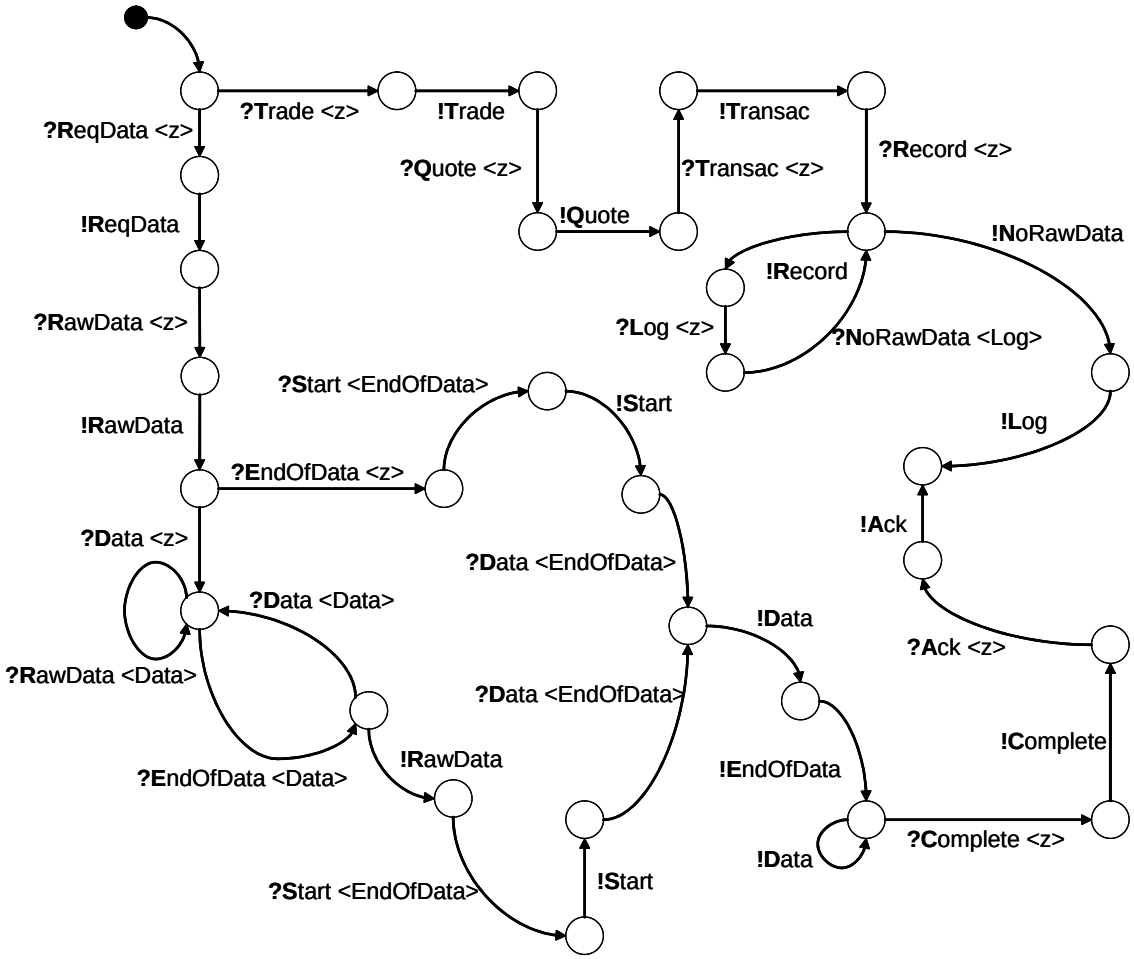


Figure 8.4: Example: generated adaptor

It should be noticed that in solving the extended FMUS service problem, we encountered a in executing MOPED. When performing pushdown model checking using MOPED, it takes much time, that is, an hour or more, in execution and finally MOPED crashed and dumped some error information. Considering the pushdown system model generated by the tool has about 128 thousands lines, we may take that the size of the model is too large for MOPED to model check on a Pentium-4 1.8G machine with 2G ram. Thus, in order to reduce the load of searching in pushdown model checking, we use a simplified LTL formula which uses only part of the Looped Transitions and Branching Transitions we found. For more details, the original LTL formula is as follows:

```
( <> ( q5_5_5_0_0_c0 && z ) ) &&
( <> pop_ReqData ) && ( <> pop_Transac ) && ( <> push_RawData ) &&
```

```
( <> push_EndOfData ) && ( <> pop_RawData ) && ( <> pop_NoRawData ) &&
( <> pop_EndOfData ) && ( <> push_Data ) && ( <> pop_Data ) &&
( <> pop_Complete ) && ( <> push_ReqData ) && ( <> push_Trade )
```

To simplify the above formula with still keeping essential information for generating an adaptor, we made the following LTL formula:

```
( <> ( q5_5_5_0_0_c0 && z ) ) &&
( <> push_RawData ) && ( <> push_Data ) &&
( <> push_ReqData ) && ( <> push_Trade )
```

Comparing to above two formulas, one may recognize that we keep the property of Behavior Mismatch Free while only keeping special symbols of pushing `RawData`, `Data`, `ReqData`, and Trade. The first two messages are the most important Unbounded Messages and the last two messages are the key messages of the two branchings in the extended FMUS service. Note that we only keep the push part of these messages in simplified LTL formula since once pushing and popping a message is coupled so if we guarantee the pushing then popping is also guaranteed. Through this simplification of LTL formula, we also learned that not all Looped Transitions and Branching Transitions are necessary for generating an adaptor. It is possible building a more compact LTL formula as well as the pushdown system model for pushdown model checking by MOPED.

Finally, we would like to summarize the evaluation of the approach based on above adaptation problem, the extended FMUS service, as well as experiments in Chapter 7. The evaluation is described with the following aspects:

- **Expressiveness of models**

  The approach proposed pushdown system model, i.e., IPS in this work, for representing adaptors with non-regular behavior. IPS is especially proposed for representing non-regular behavior from interactions of components. Interactions of components through an adaptor are basically interleaved message passings. If we monitor message passings in an adapted system, then for every message passing we should detect the message twice: once when the message is sent by a component to the adaptor, once again when the message is sent by the adaptor to a component. Usually adaptor can be expressed by finite state machines like in the conventional approach of adaptation. However, using pushdown systems model give us better ability to capture interactions of components. The push and pop operation of a stack can be directly used to simulate interactions between adaptors and components. This is especially useful when dealing with problems like the FMUS service requiring non-regular behavioral adaptors.

  On the other hand, the model representing behavior interfaces of components, IA4AD, also plays a important role in adaptor generation of the approach. Since we skip the step of designing adaptation contracts and adaptation contracts actual have information useful for adaptor generation, we need to implant the necessary information in behavior interfaces of component. The necessary information, by our observation, is the functionalities each component designated to achieve. This

is basically observed by developers and is used to design adaptation contracts. In the approach, the proposed model of components makes the information of functionalities of a component to be explicitly expressed by distinct and distinguishable initial and final states. Therefore, the approach can retrieve and use the information in adaptor generation. This also saves cost for developers by not designing a over all behavior to achieve functionalities of components but instead clearly specifying in each component. states

- **Applicability of the approach**

  In Chapter 7, demonstrations of solving adaptation problems are introduced. The first adaptation problem introduced in experiments is the FMUS service which is the motivational example of this work. The FMUS service is considered a typical adaptation problem requiring adaptors with non-regular behavior and the approach is basically designated for solving it. Then we considered general cases of adaptation problems which include signature mismatches and branchings. For signature mismatches, though the approach is not capable of dealing with signature mismatches directly, we may provide mapping components to represent mappings of labels specified by developers. Since mappings of labels are specified based on knowledge of developers, correctness of mappings are responsibilities of developers and can not be guaranteed in the approach . We only promise that given mappings of labels specified, we can introduce mapping components for representing these mappings and make the system satisfy the compatibility condition in order to perform adaptor generation by the approach. The effectiveness of introducing mapping components is confirmed using the FD service problem shown in Fig. 7.13. For branchings which is a serious problem since we use pushdown model checking to generate a counterexample, we introduce a modification of adding a epsilon transition connecting the final and initial state of the pushdown system to be checked by MOPED. Thus, exclusive branchings can be searches by pushdown model checking algorithms and the returned counterexample can reflect branchings in components. The modification for supporting branchings is also confirmed effective through the FD service problem. For further confirmation of the applicability of the approach, an extended FMUS services shown in Fig 8.1 having all issues of adaptation above is introduced for testing the applicability of the approach. The result of solving the extended FMUS service is successful and we would like to conclude that the approach proposed in this work has good applicability for adaptation problems.

- **Feasibility of the approach**

  For the experiment on FMUS service problem, we also implemented the generated adaptor as BPEL processes in Section 7.2. An adaptor is implemented as two BPEL processes for representing finite state machine part and stack part of the pushdown system model. Directions of implementing BPEL adaptor are also given in Section 7.2 as a general guidance. Thus, we proved that adaptors representing by pushdown system model in the approach is really implementable as applications. The directions for implementing BPEL adaptor also showed that the implementation is easy and does not cost much in development. Therefore, we may conclude that the approach has feasibility on real applications. Since the approach only deals with

behavior interfaces which are more abstracted than protocols in implementation such as BPEL, there are rooms for improvement of the approach to directly support adaptor generation for real applications.

Another issue about feasibility is about the execution time. It should be noted that the pushdown system model to be model checked by MOPED grows when mapping components are introduced. By examining pushdown system models for the SR service shown in Fig. 5.1, the FMUS service shown in Fig. 3.5, the lines of pushdown system model are 43 and 1923 respectively. Since the size of states in the two problems are 4 and 150, the ratio of pushdown system models is close to the size of states. Thus, in the problem of FD service shown in Fig. 7.13 with supporting mapping components shown in Fig. 7.14, lines of pushdown system model grows to 42,248. Finally, the extended FMUS service has a pushdown system model with 128,920 lines. Thus, we may conclude that scalability problem in the approach is serious if we have to introduce many mapping components.

- **Tool Support and Others**

  In this thesis, the tool implemented is only responsible for reading input and output pushdown system model for MOPED After a counterexample is returned, the tool take the responsibility to generate an adaptor from the counterexample. We may say that the tool does simple tasks in the approach. The most important part rely on tools the part of pushdown model checking by MOPED. The version of MOPED is actually an old version but is the only version supporting LTL pushdown model checking. New version of MOPED only checks reachability and is specialized for analyzing programs such as C or Java. Thus, to the subject of pushdown model checking, this old version of MOPED might need improvement on efficiency or other aspects. Therefore, some other pushdown model checkers are welcome and it would be perfect to support models like Promela in SPIN.

  Though not directly related to adaptation problems, issues about ordering of same messages in the approach should be mentioned. Since this approach uses pushdown system model, the use of stack makes the ordering of messages in the style of last-in-first-out(LIFO). Thus, messages sent multiple times such as *Unbounded Messages* have reversed ordering when being received. This might be unrealistic since services usually communicate in the style of first-in-first-out(FIFO), such as video on demand services providing streaming data. However, here we would like to point out that the ordering of *unbounded messages* in our approach can be maintained in implementation. For example, we can implement an adaptor with extra operations that adjust the ordering of *Unbounded Messages* in the stack to original ordering. We may also use another way of implementation such as building queues so that each queue corresponds to a specific messages and only store this message. As long as the implemented adaptor follows the behavior of the adaptor generated by our approach, it is irrelevant reverse or not the ordering of messages multiply sent in the implementation. Thus, we say that our approach can support both LIFO and FIFO communications practically.

Chapter 9

Related Work

The early stage of software adaptation was introduced by D.M. Yellin and R.E. Strom [21]. Their work proposed an approach of adaptation between two components which are represented by a simple description language of finite state machines. Later, Becker, et al. proposed an approach based on patterns of software adaptation [1]. As a hot topic in component-based software engineering (CBSE), many approaches are proposed for software adaptation. Most approaches, including ours, focused on solutions for behavior mismatches between abstract behavior interfaces [22, 2, 9, 10]. Brogi et al. [22] proposed an approach which uses a subset of the π-calculus for representing the behavior of components. They also used composition specifications with name correspondences for adaptor generation. This research was later improved to a more complete approach known as the model-based approach [2] which uses Labeled Transition Systems (LTSs) for modeling and calculation of software adaptation. According to our survey, this approach has defined a conventional framework for software adaptation using two basic elements: *behavior interfaces* and *adaptation contracts* which are both modeled by LTS. Some work, though using different approaches on computation, is based on this framework. Tivil et al. [9] proposed a computation technique which directly constructs partial behavior of adaptor from corresponding software components, and gives more computational efficiency to adaptor generation but incapable of solving reordering mismatches. This technique can also integrate LTL model checking by directly composition with Büchi automata transformed from specified LTL properties. Mateescu et al. [10] used process algebra for modeling behavior, LOTOS for specification of protocols, and CADP toolbox for automated on-the-fly adaptor generation.

Recently, approaches for adaptation on services became popular and techniques of software adaptation mentioned above were extended or modified for service composition. Cubo et al. applied the approach of [2] to WF/.NET framework and added verifications in their approach [3]. Mateescu et al. also extended their work in [10] to service adaptation using the model of Symbolic Transition Systems (STSs) [23]. Some other work used their own definitions for adaptation. Nezhad et al. [24] defined their own interfaces including sets of XML data and introduced an algorithm for solving interface mismatches. Mitra et al. [25] used I/O automata with history to support the multiple uses of same messages in service composition. Compare to above work, our approach attempts to capture non-regular behavior in service composition and uses pushdown automata model for representing adaptors. The use of model checking technique integrates adaptation and verification so that generated adaptor is guaranteed to satisfy both behavior mis-

match free and safety/liveness properties if specified. Our previous work [26] proposed the first version of our approach that uses Büchi automata model for behavior interfaces of services. The work proposed a property called *behavior mismatch free* defined from acceptance condition of Büchi automata. To our best knowledge, this work was the first time generation of non-regular behavioral adaptor is tackled.

Another topic in software adaptation is automated generation of adaptation contracts. For web services, it becomes a problem that adaptation contracts have to be manually specified in the conventional framework proposed in [2] while there are mobile services that demand being selected and composed dynamically. Some research about adaptation also tackles this topic in various ways. J. A. Martín and E. Pimental [20] proposed an expert system based approach which combines exploring rules and A^* graph search algorithm. Their approach automatically generates adaptation contracts (mainly mappings of labels) having the best score. Other work provides semi-automated way to guide design of adaptation contracts. Nezhad et al. [24] introduced an interactive way for users to specify adaptation contracts related to behavior mismatches on reordering. Cámara et al. developed an integrated tool ITACA [4] to support composition of BPEL services which provides interactive graphical user interface to guide the design of adaptor contracts. Compare to above work, our approach does not generate adaptation contracts but directly generates an adaptor rely on only information from behavior interfaces of services. Assuming signature mismatches are solved and mappings of labels are specified, our approach provides fully automated adaptor generation. This is proposed in our recent work [27]. Furthermore, we especially address property for *Unbounded Messages* which characterize non-regular behavior of adaptors. Though the use of model checking technique is similar to exploring graph structures of behavior interfaces, we argue that the use of model checking brings more advantages since model checking techniques are improved rapidly as well as the feature of performing both service adaptation and verifications at the same time.

Chapter 10

Conclusion and Future Work

10.1 Summary

This thesis describes the work of automated adaptor generation for behavioral mismatching components based on pushdown model checking. We focus on two major problems of conventional approach of adaptation: non-regular adaptation which needs adaptors with non-regular behavior, and skip the step of designing adaptation contracts. To solve the two problems, we introduced a motivational example, the FMUS service, to demonstrate adaptors with non-regular behavior. By following the considerations on solutions to the FMUS service, models of adaptors and components are proposed. We use pushdown system model for representing adaptors and Interface Automata model for representing components. The two models, i.e., Interface Automata for Adaptation and Interface Pushdown System, are modified with extensions as well as constraints for the purpose of adaptor generation in the approach.

Once the models in the approach are cleared, first the detection of behavior mismatches using model checking is introduced. The idea is to model checking for a property called the property of Behavior Mismatch Free which represents deadlock free for the system behavior computed by synchronously composition. Thus, we can efficiently detect behavior mismatches thanks for the state of the art model checking techniques. Furthermore, by building Promela model with parallel processes with synchronous communications, the synchronous composition can be computed by SPIN during model checking. By using model checking, the approach have another advantage that verification tasks can be integrated at the same time if safety or liveness properties are provided.

In the adaptor generation of the approach, pushdown model checking is used. First a special adaptor called Coordinator is introduced. Coordinator is designed over-behavioral so that all interleaving message exchanges through an adaptor is possible by adapted synchronous composition with Coordinator. Then pushdown model checking using MOPED is used to pick up a counterexample which is a trace for building an adaptor. This trace should have desired behavior of adaptors. Therefore, in this step pushdown model checking should check for the negation of desired property to return a counterexample suitable for generating an adaptor. The approach here do not rely on adaptation contracts but introduces a property ϕ necessary for adaptors. ϕ consists of two properties: the property of Behavior Mismatch Free and the fairness property for Looped Transitions. The latter is for capturing Unbounded Messages appearing in non-regular behavior of interactions of components. We proposed algorithms of locating Looped Transitions as well as building

an from counterexample returned by MOPED.

We have done some experiments for testing the approach. First, the FMUS service is demonstrated and showed that the approach do generate adaptors with non-regular behavior. Then BPEL implementation for the generated adaptor of the FMUS service is demonstrated to show the feasibility in software development. Directions of building BPEL adaptors are also given for references. Furthermore, two general cases of adaptation problems, i.e., signature mismatches and branchings are discusses. In order to demonstrate the two general case, another adaptation problem, the FD service is introduced. Though the approach can not solve the two general cases of adaptation directly, we have managed to solve the FD service by two modifications in adaptor generation in the approach. For signature mismatchings, we introduce mapping components for representing mappings of labels. Thus, we can still perform adaptor generation for given components with the help of mapping components. It should be noted that mapping of labels should be specified by developers and are not guaranteed by the approach. For branchings, the pushdown system model for MOPED is modified with a epsilon transition for the searching algorithms of pushdown model checking can check the behavior through multiple sessions of executions. Therefore, all exclusive branchings can be searched guaranteed to show in the returned counterexample as well as the generated adaptor. Finally, the FMUS service is extended with the two general cases and introduced as a fully mismatching adaptation problem, the extended FMUS service, to give a final test to the approach. Though the approach managed to generate an adaptor for the extended FMUS service, we encountered an scale issue and had to simplify the LTL formula manually to reduce the scale of pushdown model checking.

10.2 Contributions

This work proposed an approach that performs automated adaptor generation with only given behavior interfaces of components. The behavioral mismatching components to be composed should satisfy a few constraints such as form a closed system that all messages are mapped properly. Without design adaptation contracts in advance, this approach provides more flexibility and mobility in service composition. Also, in this approach, adaptor generation is performed by model checking so that no additional verification cost is needed. The major feature of this approach is the use of pushdown systems model. Pushdown systems are used to represent behavior interfaces of adaptors and successfully capture the nature of non-regular behavior in interactions of components. As our best knowledge, this is the first work that tackled non-regular behavior in adaptation. Furthermore, Unbounded Messages that can be sent/received arbitrary multiple times, which is the most important characteristic in non-regular behavior of service interactions, are successfully located and reflected in generated adaptors in the approach. Thanks the simplicity of the structure of pushdown systems, the approach also provides directions to implement adaptors as BPEL processes therefore feasibility for real world applications are possible. Furthermore, issues of signature mismatchings and branchings in behavior interfaces of components are discussed and modifications are proposed for solving adaptation problem having the two features. Therefore, we may conclude that the approach can generate adaptors automatically with correcting reflects both non-regular characteristics and generality in adaptation problems.

10.3 Future Work

As future directions of the approach, first we should consider improvements of issues found in the approach. As discussed in Chapter 7, general cases of adaptation, signature mismatches and branchings, need modification on the approach for solving theses cases. An improved approach of adaptor generating considering the two general cases at the start of adaptor generation is required. The input of the improved approach should consider behavior interfaces of components and mapping components when mapping of labels are specified. When branching in behavior interfaces of components are detected, the tool can automatically modify the pushdown system model as well as LTL formula for pushdown model checking. Furthermore, since there is no rule of building mapping components from specified mapping of labels, we may also need directions of building mapping components from mappings of labels. The ultimate objective is to build mapping components automatically. We may also use the help from related work on generating mappings of labels, for example, the work by J. Martín and E. Pimentel [20], to improve the ability of dealing with signature mismatches in the approach.

Another direction is in implementing applications. The approach proposed in this thesis only deal with abstracted interfaces of components. This means when we apply the approach on real applications, we need to do abstraction first. The generated adaptor is also an abstracted protocol so the adaptor generated is only an direction of implementation for real applications. Therefore, we may select a development platform, for example BPEL processes, to extend the approach on automatic adaptor generation for BPEL services. This should widen the applicability of the approach.

On the other hand, the problem of scalability we encountered in the extended FMUS service introduced in Chapter 8 where the pushdown system model to be model is too large and there are too many transitions added as fairness in the LTL formula is also vital to the approach. Though we have managed to simplify the LTL formula by eliminating some fairness properties pf transitions in LTL formula, a systemic way of doing such kind of reduction is required. We should develop a process of reducing the scale of pushdown system model though eliminating unimportant transitions in Looped Transitions and Branching Transitions. Therefore, we can build more compact pushdown system model since special stack symbols are reduced as well as corresponding transition rules. Furthermore, we may try another way of dealing with the size of pushdown system model by changing the input model to the Remopla, the new input model of MOPED. Thus, optimizations in implementing MOPED may also help us reducing the size of pushdown system model.

Bibliography

[1] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an engineering approach to component adaptation. In *Architecting Systems with Trustworthy Components*, pages 193–215, 2004.

[2] Carlos Canal, Pascal Poizat, and Gwen Salaün. Model-based adaptation of behavioral mismatching components. *IEEE Trans. Softw. Eng.*, 34(4):546–563, 2008.

[3] Javier Cubo, Gwen Salaün, Carlos Canal, Ernesto Pimentel, and Pascal Poizat. A model-based approach to the verification and adaptation of wf/.net components. *Electron. Notes Theor. Comput. Sci.*, 215:39–55, 2008.

[4] Javier Camara, Jose Antonio Martin, Gwen Salaun, Javier Cubo, Meriem Ouederni, Carlos Canal, and Ernesto Pimentel. Itaca: An integrated toolbox for the automatic composition and adaptation of web services. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 627–630, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Javier Cámara, Gwen Salaün, and Carlos Canal. Clint: a composition language interpreter (tool paper). In *Proceedings of the Theory and practice of software, 11th international conference on Fundamental approaches to software engineering*, FASE'08/ETAPS'08, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[7] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[8] Steffen Becker, Sven Overhage, and Ralf H. Reussner. Classifying software component interoperability errors to support component adaption. In *Component-Based Software Engineering, 7th International Symposium, CBSE 2004*, pages 68–83. Springer, 2004.

[9] Massimo Tivoli and Paola Inverardi. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program.*, 71(3):181–212, 2008.

[10] Radu Mateescu, Pascal Poizat, and Gwen Salaün. Behavioral adaptation of component compositions based on process algebra encodings. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 385–388, New York, NY, USA, 2007. ACM.

[11] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[12] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, JUN 1986.

[13] Javier Esparza and Stefan Schwoon. A bdd-based model checker for recursive programs. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 324–336, London, UK, 2001. Springer-Verlag.

[14] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.

[15] Xiang Fu, Tevfik Bultan, and Jianwen Su. Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theor. Comput. Sci.*, 328(1-2):19–37, 2004.

[16] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[17] Graphviz - graph visualization software. http://www.graphviz.org/.

[18] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web service definition language (wsdl). Technical Report NOTE-wsdl-20010315, World Wide Web Consortium, March 2001.

[19] ORACLE. Netbeans ide. http://netbeans.org/.

[20] José Antonio Martín and Ernesto Pimentel. Automatic generation of adaptation contracts. *Electron. Notes Theor. Comput. Sci.*, 229(2):115–131, 2009.

[21] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

[22] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74(1):45–54, 2005.

[23] Radu Mateescu, Pascal Poizat, and Gwen Salaün. Adaptation of service protocols using process algebra and on-the-fly reduction techniques. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 84–99, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] Hamid Reza Motahari Nezhad, Boualem Benatallah, Axel Martens, Francisco Curbera, and Fabio Casati. Semi-automated adaptation of service interactions. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 993–1002, New York, NY, USA, 2007. ACM.

[25] Saayan Mitra, Ratnesh Kumar, and Samik Basu. Automated choreographer synthesis for web services composition using i/o automata. In *2007 IEEE International Conference on Web Services (ICWS 2007)*, pages 364–371, 2007.

[26] Hsin-Hung Lin, Toshiaki Aoki, and Takuya Katayama. Non-regular adaptation of services using model checking. In *ISORC '10: Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 170–174, Washington, DC, USA, 2010. IEEE Computer Society.

[27] Hsin-Hung Lin, Toshiaki Aoki, and Takuya Katayama. Automated adaptor generation for services based on pushdown model checking. In *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on*, pages 130 –139, april 2011.

Publications

[1] Hsin-Hung Lin, Toshiaki Aoki, and Takuya Katayama. Automated adaptor generation for services based on pushdown model checking. In *ECBS2011: 18th IEEE International Conference and Workshops on Engineering of Computer Based Systems*, pages 130 –139, april 2011.

[2] Hsin-Hung Lin, Toshiaki Aoki, and Takuya Katayama. Non-regular adaptation of services using model checking. In *ISORC '10: Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 170–174, Washington, DC, USA, 2010. IEEE Computer Society.

[3] Hsin-Hung Lin and Takuya Katayama, "Towards Integrating Adaptation and Model Checking for Software Components," IPSJ SIG Tech. Rep., Vol. 2009-SE-165, 2009.

[4] Hsin-Hung Lin and Takuya Katayama, "Coordination and Verfication of Software Components Orchestrated by Coordinator," 14th JSSST Workshop on Foundation of Software Engineering (JSSST FOSE2007), pp.173-178, 2007.

[5] Hsin-Hung Lin and Takuya Katayama, "Communication Model Among Statecharts: an Approach Using Characteristic Event Sequences," IEICE Tech. Rep., vol. 105, no. 332, SS2005-53, pp. 31-36, Oct. 2005.