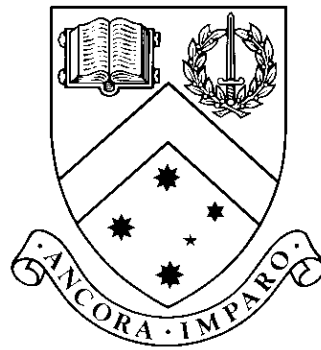


# Component-based Adapter Generation

by

Yin San Phoon



**Thesis**

Submitted by Yin San Phoon

in partial fulfillment of the Requirements for the Degree of  
**Bachelor of Software Engineering with Honours (2770)**  
in the School of Computer Science and Software Engineering at  
Monash University

**Monash University**

November, 2003

© Copyright

by

Yin San Phoon

2003

# Contents

<b>List of Figures</b> . . . . .	<b>v</b>
<b>Abstract</b> . . . . .	<b>vi</b>
<b>Acknowledgments</b> . . . . .	<b>viii</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
<b>2 Architectural Description</b> . . . . .	<b>3</b>
2.1 KOALA . . . . .	3
2.2 Architecture Reuse . . . . .	4
<b>3 Component Adaptation</b> . . . . .	<b>5</b>
3.1 Motivation for Adaptation . . . . .	5
3.2 Adaptation by Generating Adapters . . . . .	6
<b>4 Adapter Generation and Protocol Specification</b> . . . . .	<b>7</b>
4.1 Process Algebra . . . . .	7
4.2 Finite State Machine . . . . .	8
<b>5 Systematic Method of Adapter Generation</b> . . . . .	<b>10</b>
5.1 Architectural Definition of Components . . . . .	10
5.2 Adapter Automation . . . . .	12
5.2.1 Step 1: Specify . . . . .	12
5.2.2 Step 2: Generate . . . . .	14
5.2.3 Step 3: Verify . . . . .	15

<b>6</b>	<b>Implementation, Results and Discussion</b>	<b>18</b>
6.1	Preparation of case study	18
6.2	Realization of <i>Adapter Automation</i>	20
6.3	Example from case study	22
6.3.1	Step 1: Specify	22
6.3.2	Step 2: Generate	23
6.3.3	Step 3: Verify	25
6.4	Discussion	28
<b>7</b>	<b>Conclusion</b>	<b>30</b>
	<b>References</b>	<b>32</b>
	<b>Diagrams</b>	<b>34</b>

# List of Figures

5.1	Architectural view . . . . .	11
5.2	Abstract behaviour of <code>dvd_control</code> modelled by the FSM in (b) . . . . .	13
5.3	The 3 types of adapters . . . . .	15
5.4	Types of connection between 2 gates . . . . .	16
6.1	Graphical notation of a FSM for <code>record_vcr</code> generated using the <code>graphviz</code> tool	20
6.2	Architecture design of the DVD component . . . . .	20
6.3	DVD ken . . . . .	22
6.4	A 1:2 adapter is inserted . . . . .	23
6.5	FSM in (c) shows all the possible interleavings of FSM in (a) and (b) . . . .	24
1	Architecture design of a DVD_VCR player. See previous page for the regular expressions of the main components. . . . .	35
2	FSMs <b>before</b> correctness checking of <code>record_vcr</code> . . . . .	36
3	FSMs returned <b>after</b> the correctness checking. Error is revealed to be located in the Start state. This diagram suggests to the designer that, the start state is also the initial state, may be the correct design. . . . .	37

# Component-based Adapter Generation

Yin San Phoon, BSE(Hons)  
Monash University, 2003

Supervisor: Prof. Heinz Schmidt

## Abstract

In Component-Based Software Engineering, software components can rarely be reused without any adaptation. Ideally, pre-existing software components can be interconnected and made to cooperate to perform the required tasks. But, achieving interoperability between the components is very difficult since the heterogeneity of the components causes unforeseen incompatibilities to arise when the components are being connected together. Therefore, the software components need to be adapted. Recent approaches in component adaptation have been geared towards adapter generation. However, insufficient information on component interface presents a major obstacle in this approach. The lack of interface protocol information causes the protocol incompatibilities of components to be unresolved.

The main concern of this research is to devise a systematic method in adapter generation in order to resolve the protocol incompatibilities. This method adopts architectural definition of components and formal specification of protocol using Finite State Machines in order to systematically generate adapter to remove incompatibilities. This method is realized by building an extension to an existing FSA Package as Java library. This thesis presents in detail the method that has been devised for adapter generation. This method supports automatic and user-driven adapter generation depending on the behaviour of the components.

# Component-based Adapter Generation

## Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

---

Yin San Phoon  
November 11, 2003

# Acknowledgments

I would like to thank my supervisor and colleagues for all the help and support they provided throughout this year. Your help has allowed me to complete this research project and to pull through a very tough year.

Yin San Phoon

*Monash University*  
*November 2003*

# Chapter 1

## Introduction

In the context of architecture reuse, reusing pre-existing software components straight off is very desirable. But, in reality components can rarely be reused without any adaptation. Software components are designed by different people, for certain purposes and to work under certain environment. Due to their heterogeneity, unforeseen incompatibilities, where components are unable to fulfill all the interoperability requirements, may arise when the components are being interconnected (Vallecillo, Hernandez and Troya, 2000). As a result, a designer needs to perform adaptation to enable the components to interoperate. Rather than modifying the existing components, a better way of adaptation is to generate adapter that will reside in between the components.

A good understanding of the structure and how the components interoperate is necessary in order to generate adapter. Therefore, approaches in architecture description and protocol specification were analysed. The architectural description provides the architectural view of a system, which is composed of components with connectors linking them together (Jazayeri, Ran and der Linden, 2000). It can also provide information on the behaviour of the components (protocol specification).

A few approaches can be used to model the specification of the abstract behaviour and the interfaces of the components. One approach to model the protocol specification is using process algebra (Bracciali, Brogi and Canal, n.d.; Allen and Garlan, 1997). Bracciali et al. (n.d.) presented a formal methodology in adapting incompatible components formally using process algebra. The usage of process algebra promotes the descriptiveness of the protocol. However, its complexity in analysis causes a major setback. Due to the complexity of the analysis, the verification of adapter becomes more time consuming as the system becomes more complex. An alternative approach in modelling protocol specification is using the Finite State Machines (FSM) (Yellin and Strom, 1995; Schmidt and Reussner, 2002a; Schmidt and Reussner, 2002b; de Alfaro and Henzinger, 2001; Passerone, de Alfaro, Henzinger and

Sangiovanni-Vincentelli, 2002). The simplicity of FSM in modelling the protocol specification supports a simple and therefore an efficient verification of protocol compatibility.

This research focuses on devising a systematic method for generating adapters to resolve behavioural incompatibilities through architecture description and by modelling the protocol specification of interfaces with FSM. The architectural description and the algorithms for generating adapters are based on Schmidt and Reussner's (2002b) paper whereas Bracciali et al.'s (n.d.) paper is used as the main reference for the steps devised for this method. For this research, a DVD\_VCR player is used as a case study. The method devised is implemented as an extension to an existing FSA package built in Java.

Chapter 2, 3 and 4 discusses the relevant literature on architectural description, component adaptation and especially in adapter generation, and protocol specification. Chapter 5 presents the concept of the devised method for adapter generation and Chapter 6 describes the implementation, results and discussion of that method based on some examples of a DVD\_VCR player. Finally, Chapter 7 presents the conclusion for this thesis.

## Chapter 2

# Architectural Description

Architectural description holds a particular importance in documenting and conveying the views of the architecture of a system (Jazayeri et al., 2000). It consists of 2 parts, namely the notation and the language. The architectural view, which can be seen as a type of abstraction, usually contains a collection of components together with a collection of connectors, which describe the interaction of the components. Adapter, as discussed in the previous section, belongs to the realm of connectors according to Marangozov, Bellissard, Vion-Dury and Riveill (1997). The abstraction of the architectural view provides a better understanding for a large and complex system.

### 2.1 KOALA

There are many Architecture Description Language(ADL) that exist nowadays to help the software architect to gain a better understanding on the architecture of a system (Jazayeri et al., 2000). However, the discussion from here onwards will only focus on Koala (Jazayeri et al., 2000), which is an ADL that is based on Darwin. It is developed specifically by the ARES project from embedded software for product families. Just like Darwin, Koala provides architectural view of the hierarchical decomposition of component. In other words, it shows the primitive components as well as the composite components of a system. A composite component is made up of a few primitive components. A benefit of Koala is that it places greater emphasis in deployment architecture. Another advantage of Koala is that it distinguishes a component type from component instance, allowing software architect to define and analyze the component independent of their actual use. The component can then be instantiated for a particular reuse. The notation used in describing the architecture also proves to be beneficial. Koala represents component as a rectangle box and the interfaces as small boxes containing black triangles. The direction of the triangles indicates intuitively the direction of function calls. The KOALA concept of architecture description is closely related to the one described in Schmidt and Reussner's (2002b) paper and latter one is

being adopted in this research. The description of the latter will be presented in Section 5.1.

## 2.2 Architecture Reuse

In the context of architecture reuse, existing components can rarely be reused without any adaptation. Therefore, research in the area of component adaptation has become increasingly popular over the years. The details of the motivation for component adaptation and the approaches taken will be discussed in the next chapter.

Formalizing software architecture specification is necessary in order to perform adaptation. According to Allen and Garlan (1997), there is a need to formally specify software architecture, specifically the interactions between architecture components. They emphasized in their paper that there is not much use if a software designer only has an informal description of the architecture of a system, accompanied by a diagram showing boxes connected with arrows without the knowledge to reason the interaction between the components. Due to the prevention of analyzing the interaction between the components, the designer will not be able to fix incompatibilities between components by adaptation without going into the implementation of the component. Allen and Garlan proposed the enhancement of the architectural description language (ADL) by modelling the behaviours of interacting protocols using process algebra. A more detailed discussion on modelling the protocols will be presented in Chapter 4.

## Chapter 3

# Component Adaptation

In recent years, software component adaptation has been a popular field of research. Software components are designed by different people, for certain purposes and to work under certain environment. Due to their heterogeneity, unforeseen incompatibilities, where components are unable to fulfill all the interoperability requirements, may arise when the components are deployed (Vallecillo et al., 2000). Therefore, components can rarely be reused as they are without any adaptation.

### 3.1 Motivation for Adaptation

The incompatibilities that occur may be caused by low-level interoperability problems like the incompatibilities in programming languages, operating platforms or database schemes (Garlan, Allen and Ockerbloom, 1995). But, this research is focusing on dealing with the high-level interoperability problems or referred to as architectural mismatches, more specifically, mismatch in the interaction protocols between component interfaces. A more detailed explanation in architectural mismatches can be found in Shaw's (1995) paper and Garlan et al.'s (1995) paper. The latter paper highlighted the architectural mismatches using an example in building a family of software design environment from existing parts.

The need for adaptation of software components is also explained in Heineman's (1999) paper. According to him, integrating a third-party software into an existing system may be difficult due to syntactic incompatibilities between the interfaces, which act as communicators for the system and the component. In other words, the system interface and the component interface have to be compatible with each other in order for them to communicate with one another. In his paper also, he states that in order to overcome the syntactic incompatibilities, either the existing system may be modified or the component may be modified by introducing a component adapter.

## 3.2 Adaptation by Generating Adapters

Introduction of a component adapter is the most suitable since modification of an existing component or system is expensive and time consuming. This is mainly because the application builder needs to understand the underlying source codes before hand. Another factor that shows the suitability of generating adapter is due to the fact that apart from the syntactic problem, incompatibilities may arise as well due to the unpredictable behaviour of the components (Heineman, 1999). A component designer cannot foresee all possible behaviour of the component (Schmidt and Reussner, 2002a). Therefore, a mechanism, which will allow the application builder to adapt a component without knowing the source code will be invaluable. In order to generate adapter, information on the behaviour of the components is necessary. However, protocol specification is vital in order to generate adapter. So, the next section describes the approaches taken in adapter generation, which is garnered towards formal protocol specifications.

## Chapter 4

# Adapter Generation and Protocol Specification

To start the discussion with the importance of protocol specification, a brief explanation on the reason for extending the traditional interface definition language is necessary. As mentioned in the previous section, components interact with one another through their interfaces. Interface can be referred to as an abstraction of the interaction description of a component (Bracciali et al., n.d.). The traditional interface definition language (IDL), like CORBA, contains only the signature definition, which is the description of the functions or services provided and required by the component. The IDL lacks the description of the behaviour or known as the protocol specification of a component. Yellin and Strom's (1995) work has shown that extending traditional IDL with protocol specification for component interface has resulted in semi-automatic generation of adapter. It can be concluded that without the protocol specification, synchronization of method calls from the component interfaces cannot be achieved. Therefore, protocol specification is vital in the generation of adapter to bridge the incompatibilities of components.

### 4.1 Process Algebra

One approach for protocol specification of interface is using process algebra (Bracciali et al., n.d.; Allen and Garlan, 1997). Bracciali et al. (n.d.) presented a formal methodology in adapting incompatible components formally using process algebra. Their methodology consists of 4 main steps, namely component interfaces specification, adapter specification, adapter derivation and verification of adapter properties. Firstly, the behaviour of the interfaces is expressed using process algebra during component interfaces specification. Then, the adapter that will mediate the interoperation of the two components with incompatible behaviours is specified by a set of mapping between actions and parameters of the two components. After the adapter specification, a concrete adapter can be automatically generated

by an extended algorithm for checking the correctness of an open context of components using the protocols specification and the mapping established earlier. Once the adapter is generated, each rule of the mappings or in other words the properties that the adapter must satisfy will be validated. This is achieved by projecting the adapter over the actions which are relevant to the property in concern. On the whole, this paper certainly presents a very formal method of generating adapter for incompatible components. Although using process algebra for protocol specification contributes to the expressiveness of description of the protocol, it does have a major draw back due to the complexity in performing analysis. The complexity in analysis will eventually cause the verification of adapter to be more time consuming as the system becomes more complex.

## 4.2 Finite State Machine

An alternative approach is using Finite State Machines (FSM) (Yellin and Strom, 1995; Schmidt and Reussner, 2002a; Schmidt and Reussner, 2002b; de Alfaro and Henzinger, 2001; Passerone et al., 2002). As oppose to using process algebra, the simplicity of FSM in representing protocols of the interfaces supports a simple and efficient verification of protocol compatibility. By protocol compatibility, it means the required services of a component must be provided by the component it is interacting with. In addition, they must agree also on the sequences of the message received and sent (Yellin and Strom, 1995).

Yellin and Strom's (1995) approach in specifying the protocol of interfaces using FSM has resulted in the semi-automatic generation of adapter to bridge incompatibilities between components. They adopt the notion of protocol compatibility from On Communication FSM (Brand and Zafropulo, 1983), where FSM is a run time device to control client server interaction. Yellin and Strom specify the protocols for the required and the provided services in a single FSM. Thus, enabling the adapter generated to bridge incompatibilities for alternating interacting components. A draw back in their research is that it does not support the generation of adapters for concurrent system.

On the contrary, Schmidt and Reussner's (2002a) paper deals with adapter generation for concurrent system specifically. In their approach, FSMs for the provided and required interfaces of a component are modeled separately. Schmidt and Reussner have identified 3 common situations, which protocol incompatibilities may arise. For instance, when one component uses 2 other components, when 2 components use the same component concurrently and lastly when one component uses another component that is not compatible. In their paper, they show how synchronization of protocols for the 3 cases of incompatibilities can be achieved. The synchronization is done by merging the interfaces in concern into a single required or provided interface (depending on the situation mentioned just now) using shuffle FSM construction and thus generating an adapter semi-automatically. For example, when one component uses 2 components, the provided interfaces of the 2 components are

merged into one provided interface and consequently generating a split adapter that acts like a switch. Therefore, this is a very effective approach for adapter generation especially for concurrent system since the protocol synchronization of concurrent usage of component is supported. Another benefit is that the simplicity of FSM allows simple and efficient protocol compatibility verification, as mentioned before. Furthermore, dependency analysis is supported since the all of the required and provided interfaces of a component have their own model of FSM, allowing the projection of sequence calls of the interfaces in concern.

Therefore, from the analysis of the relevant literature, as shown from the discussion in the previous two chapters and this chapter, architecture description and formal protocol specification through FSM are the main aspects in generating adapter to resolve behavioural incompatibilities.

## Chapter 5

# Systematic Method of Adapter Generation

The aim of this research is to devise a systematic method for generating adapter in order to resolve behavioural incompatibilities. This chapter describes only the concept of **Adapter Automation**, which is the method that has been devised, and the implementation, results and discussions of this method will only be presented in the next chapter.

The first section in this chapter describes the architecture notation used in this research. Then, the procedures, which **Adapter Automation**, is comprised of, will be explained in the second section.

A DVD\_VCR player is used as the case study in this research. Therefore, examples used during the explanation are extracted from the implementation of the DVD\_VCR case study.

### 5.1 Architectural Definition of Components

The architecture description for this research is based on Schmidt and Reussner's (2002b) paper. Their description provides a hierarchical view of the components of a system. The view shows constructs of a component, its interfaces as well as its connections to other components via the interfaces as shown in the Figure 5.1.

A ken is used to describe a component with its extra specifications. For the purpose of this research, the specification required is the protocol specifications. Each ken consists of an *Abstract Behaviour*, *Provided Interfaces* and *Required Interfaces*. Ken can be divided into 2 types; primitive ken and composite ken. Primitive ken is the lowest level ken from an architectural point of view. As of the composite ken, it can be composed from sub-kens that are wired together or represented using regular expressions. For example, A+B, where

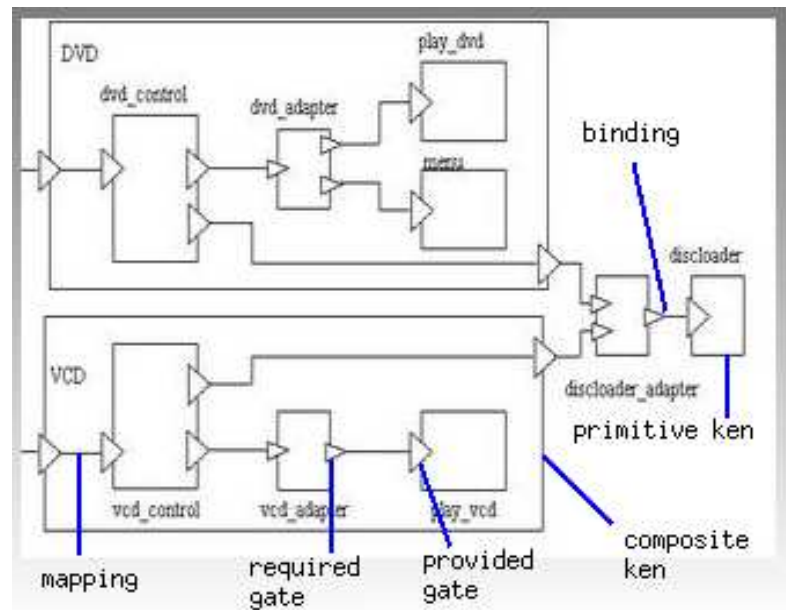


Figure 5.1: Architectural view

both A and B represents a Finite State Machine(FSM).

A gate is used to represent an interface of a component. It provides information on how to access the ken and how the ken interoperates. Gate is divided into provided and required gate. Provided gate provides information on the services provided by the ken whereas required gate provides information on the services required by the ken.

Connection between the gate is also categorized into two types, namely *binding* and *mapping*. A binding refers to the connection between a required and a provided gate while a mapping refers to the connection between an interior and exterior gate. A *provided mapping* refers to the connection between the interior and exterior provided gate whereas a *required mapping* refers to the connection between the interior and exterior required gate.

The following section will present in detail the **Adapter Automation** method with regards to the architecture description explained above.

## 5.2 Adapter Automation

Adapter Automation consists of 3 main steps, namely the *Specify*, *Generate* and *Verify*.

1. *Step 1: Specifying behaviour of gates*

In this step, the software designer specifies the protocol of gates as well as the abstract behaviour of the ken. For interfaces that will be connected, designer distinguishes the compatible and incompatible connections.

2. *Step 2: Generating adapter*

For connection that is incompatible, adapter is generated based on the incompatibilities identified in the previous step. The adapter generation can be either automatic or user-driven.

3. *Step 3: Verifying compatibility of adapter*

After adapter has been generated, its compatibility will be verified.

The following subsections explain the 3 steps of Adapter Automation in detail.

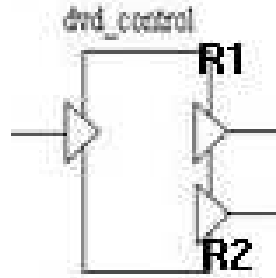
### 5.2.1 Step 1: Specify

Based on the architecture design of a system, the designer specifies the abstract behaviour of a ken as well as the protocol for its provided and gates. FSM is used to individually model the abstract behaviour, provided gates and required gates of a ken. For example, the abstract behaviour of the `dvd_control` in Figure 5.2(a) is modelled by the the FSM in Figure 5.2(b) whereas gate **R1** and **R2** are modelled by their own FSMs. The FSM describes the signatures list as well as the protocol specification of the gates. Mathematically, a FSM  $F$  can be expressed as  $P = (I, S, s_0, F, T)$  where

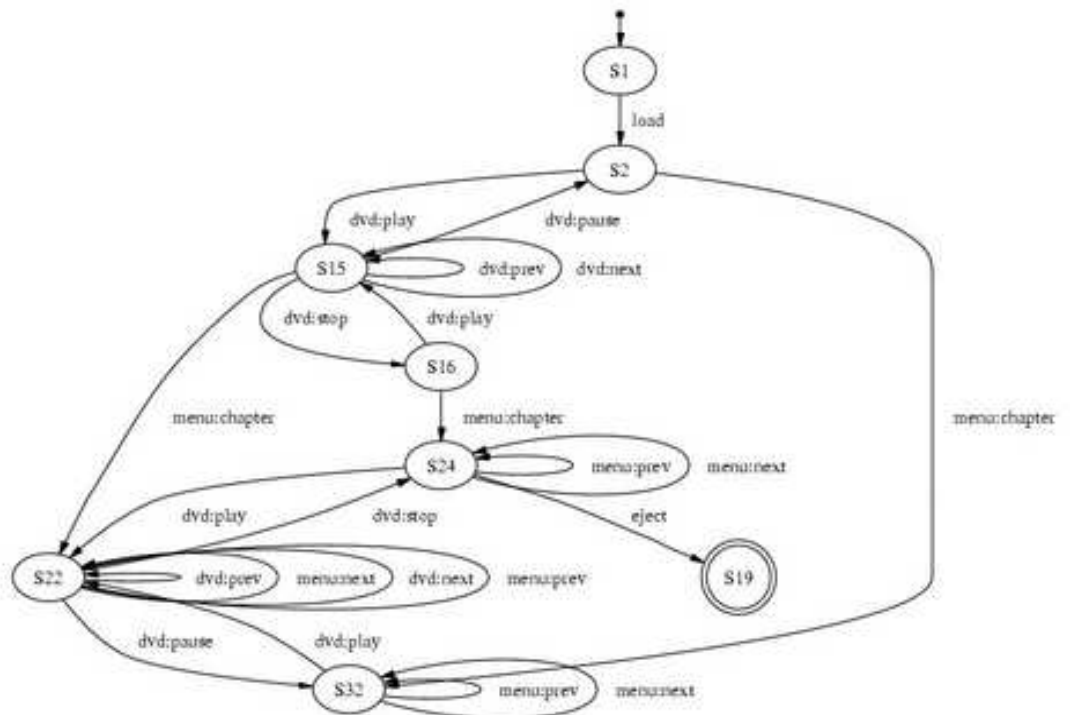
- $I$  is a set of input alphabets
- $S$  is a set of states
- $s_0$  is the initial state which satisfies  $s_0 \in S$
- $F$  is a set of final states which satisfies  $F \subseteq S$  and
- $T$  is a set of transitions from the total transition function  $t$  where  $t : S \times I \rightarrow S$

The FSM for a provided gate describes the protocol of the services provided by the ken whereas the FSM for a required gate describes the protocol of the services that the ken needs (Schmidt and Reussner, 2002b).

Automata operations can be used during the specification. Some common automata operations include concatenation ( $\bullet$ ), intersection ( $\cap$ ), union ( $\cup$ ), shuffle-product ( $\times$ ) and



(a) dvd\_control ken



(b) R\_playback FSM

Figure 5.2: Abstract behaviour of dvd\_control modelled by the FSM in (b)

iteration (\*). For example, the protocol of a ken *dvd\_control* is expressed as the regular expression  $dvd\_control = play\_dvd \times menu$ , where *play\_dvd*, *menu* and *dvd\_control* are all FSMs.

Then, for all the gates that have been specified, their correctness need to checked using the following rules:

- $L(R') \subseteq L(R)$ , if R is a **required gate**,  
where  $L(R')$  and  $L(R)$  are the regular language for  $R'$  and  $R$ , and  $R'$  is the projection of  $R$  over the abstract behaviour of its ken
- $L(P) \subseteq L(P')$ , if P is a **provided gate**,  
where  $L(P)$  and  $L(P')$  are the regular language for  $P$  and  $P'$ , and  $P'$  is the projection of  $P$  over the abstract behaviour of its ken

For example, to check the correctness of  $R1$  gate of ken *dvd\_control* (see Figure 5.2(a)),

$$\begin{aligned} A &:= \text{Abstract Behaviour of } dvd\_control \text{ ken} \\ R1' &:= (A) \setminus R1 \quad (\text{projection of } R1 \text{ over } A) \end{aligned}$$

Since  $R1$  is a required gate, then first rule is applied.  $R1$  is correct if  $L(R1') \subseteq L(R1)$  is satisfied. Intuitively speaking, *every* call sequences generated by  $R1'$  FSM is accepted by the  $R1$  FSM.

### 5.2.2 Step 2: Generate

The type of adapter generated depends on the type of incompatibilities that occurs. This research focuses on 3 types of adapters (see Figure 5.3). They are as follows:

1. *1:2 Adapter*  
This adapter is generated for resolving incompatibility that arises when two kens are used by another ken. This adapter receives an incoming call and dispatches it to the appropriate ken. In other word, this adapter functions like a split operator.
2. *2:1 Adapter*  
This adapter is generated for resolving incompatibility that arises when one ken needs to use 2 different ken simultaneously. This adapter synchronizes the calls from the 2 calling kens to the other ken. In other word, this adapter functions like a join operator.
3. *1:1 Adapter*  
This adapter is generated for resolving incompatibility that arises when one ken uses another ken. The adapter in this case functions as the protocol changer.

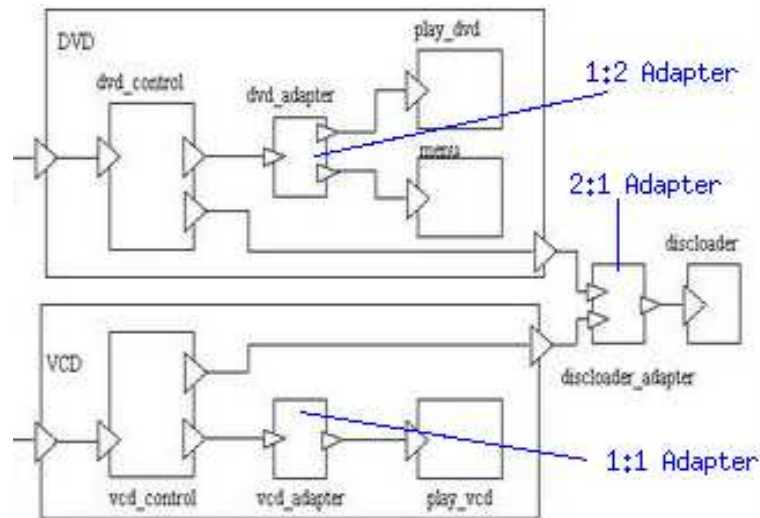


Figure 5.3: The 3 types of adapters

The detailed description and the algorithm used to generate these adapters can be found in Schmidt and Reussner's (2002b) paper.

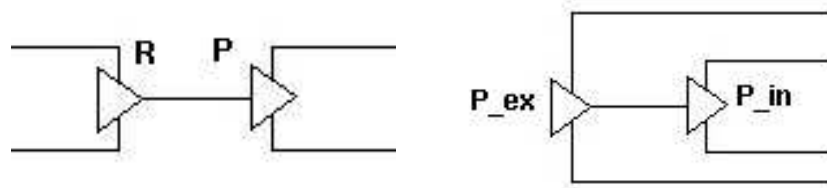
In this step, adapter is generated by applying automata operations on the specified FSMs. The main operation used for adapter generation is shuffle-product ( $\times$ ). The resulting FSM that is constructed using shuffle-product shows all the possible interleavings of the 2 FSMs specified. Note that it is due to the interleavings that allows the simultaneous usage of kens. If the designer requires only some interleavings of the 2 FSMs, then, the designer may perform further restriction on the resulting FSM for the adapter by projecting the desired behaviour (which is specified by the designer) over the resulting FSM. Not all adapter can be generated automatically since not all incompatibilities can be removed automatically. Therefore, designer may need to apply further automata operations to the behaviour of the adapter.

### 5.2.3 Step 3: Verify

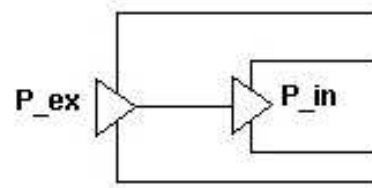
After the adapter has been generated and all the necessary restrictions have been performed, the adapter needs to be verified for its compatibility. If the adapter is not compatible with its environment, either an error message is displayed or the designer is prompted for further operation to be performed.

A component is said to be compatible with its environment if its contract *conforms* to the architectural context and its implementation is *correct* (Schmidt and Reussner, 2002a).

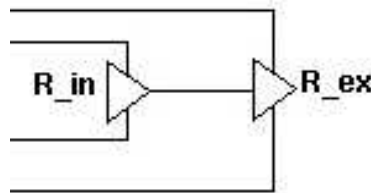
*Correctness* refers to the relationship between the interface and implementation of a component in terms of consistency and completeness. So, in an architectural context, it means the protocol specifications of a gate is implemented completely by the ken and the implementation is consistent to the protocol specifications. On the other hand, *conformance* refers to the relationship between 2 interfaces of different components. So, in an architectural context, it refers to conformance of a binding or a mapping. A binding or mapping conforms if *every* call sequences generated by the gate that requires the service is accepted by the gate that provide the service (refer to rules below).



(a) Binding



(b) Provided Mapping



(c) Required Mapping

Figure 5.4: Types of connection between 2 gates

The same rules for correctness checking in the *Specify* step is the same for this step. As for the conformance checking, the 3 rules to apply are:

- $L(R) \subseteq L(P)$ , if connection is a **binding**, where  $L(R)$  and  $L(P)$  are the regular languages for required and provided gate respectively,
- $L(P_{ex}) \subseteq L(P_{in})$ , if connection is a **provided mapping**, where  $L(P_{ex})$  and  $L(P_{in})$  are the regular languages for exterior and interior provided gates respectively, and

- $L(R_{in}) \subseteq L(R_{ex})$ , if connection is a **required mapping** where  $L(R_{in})$  and  $L(R_{ex})$  are the regular languages for interior and exterior required gates respectively

So, an adapter is compatible only if

1. All its provided gates and required gates are correct, and
2. All its bindings and mappings conforms

If an adapter is not compatible, an error message is displayed and user may be prompted for further operations in order to resolve the incompatibilities.

## Chapter 6

# Implementation, Results and Discussion

The implementation and experiments for this research can be divided into 3 main aspects; preparation of the case study for the DVD\_VCR player, develop functions needed to realize the concept of **Adapter Automation** and experimenting on the components of the DVD\_VCR player.

In order to specify the abstract behaviour and the protocol of the gates of a ken systematically, they are expressed as regular expression that is either comprise of a single FSM or a combination of FSM. Regular expression is treated like a function.

### 6.1 Preparation of case study

First of all, the behaviour of an actual DVD\_VCR player was analyzed. The operations of the DVD\_VCR player were then separated into a few main components and was expressed a regular expression. Then, the components were broken down further into regular expressions until a component that was small and simple enough to be easily specified in one single FSM (specified in a text file). This step can be viewed as the top down recursive break down of the DVD\_VCR player.

To put it in a formal way, a component and the regular expression that defines it, are treated as the left hand side and right hand side of an equation respectively. Therefore, each element of the regular expression can be a variable. The variable is either substituted by a FSM from text file or by a FSM computed from another equation. For example, in

$$DVD\_VCR = power \bullet (play\_vcr\_only + (record\_vcr \times dvd) + vcd)$$

Variable *power* is substituted by a FSM specified in a text file while the rest of the variables are FSMs computed by the following equations:

$$\begin{aligned} play\_vcr\_only &= tapeloader \bullet play\_vcr \\ (record\_vcr \times dvd) &= (tapeloader \bullet record\_vcr) \times (discloder \bullet (menu \times play\_dvd)) \\ vcd &= discloder \bullet (play\_vcd) \end{aligned}$$

A text file for specifying a FSM is shown as below:

```
(record_vcr.txt)

0 1 rvcr:channel
1 1 rvcr:speed
1 2 rvcr:record
2 3 rvcr:stop
3 2 rvcr:record
2 4 rvcr:pause
4 2 rvcr:record
3
```

Each line represents a transition. The first column is the source state, the second column is the destination state while the third column is for the input symbol. 0 is always the start state and the final state is always the entry with only the first column stated. There can be more than one final states.

The AT&T FSM package were used to compile the text files into FSM and graphviz tool is used to convert FSM into graphical notation (see Figure 6.1).

Based on the regular expressions that were derived, the architecture design of the DVD\_VCR player was created using the concept of kens, gates, binding and mapping as explained earlier. This research is focusing on adapter generation for the 3 types of adapters described previously. Therefore, when creating the design, it was always being bear in mind to create some instances in the design where these adapters were needed. Refer to Figure 1 in Appendix for the whole architecture design of the DVD\_VCR player. Adapters are included in the design.

After the architecture design of the system is completed, changes were required in the specification. Hence, the regular expressions and specification text files were either modified or new ones were created to specify the abstract behaviour of the kens and the protocol of the

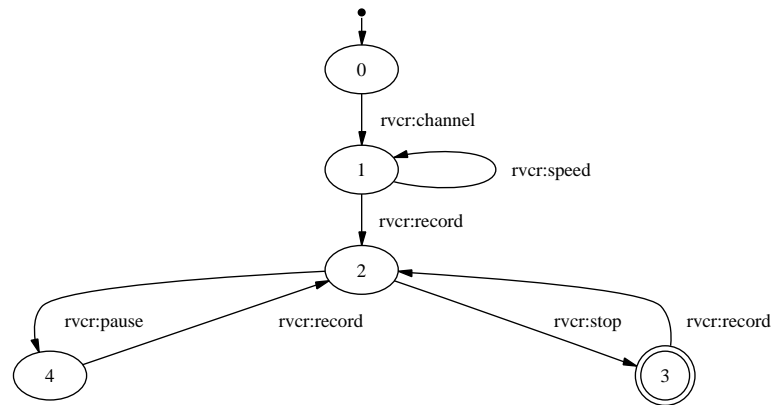


Figure 6.1: Graphical notation of a FSM for record\_vcr generated using the graphviz tool

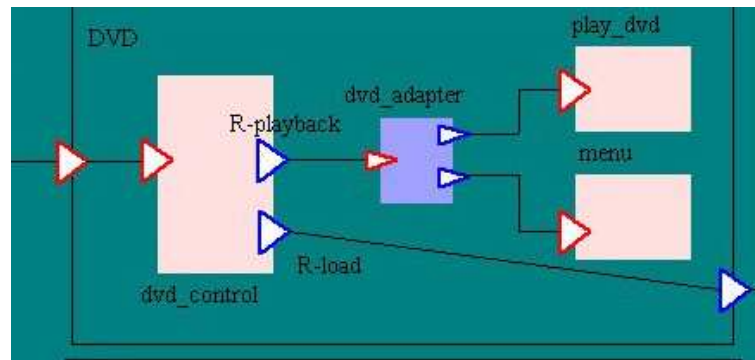


Figure 6.2: Architecture design of the DVD component

gates in order to reflect the design.

With the architecture design and all the specification (the list of regular expressions and text files) for the DVD\_VCR case study in place, the preparation of the data for the DVD\_VCR case study is completed. The research were ready to proceed to the implementation and experimentation of the *Adapter Automation* concept using these data.

## 6.2 Realization of *Adapter Automation*

Initially, scripts were created using the AT&T FSM package to implement the 3 steps in *Adapter Automation* and to test the DVD\_VCR case study. The AT&T package provides

functions for applying automata operations on the FSM. For example, concatenation, iteration, union, projection, product and so on. However, it was found that using this package for the implementation is not feasible. This is due to the fact that the specification of some of the functions for automata operations provided by this package differs from what is required to implement the 3 steps in *Adapter Automation*. Consequently, these functions were not able to perform as expected. For example, the projection in this package filters FSM that contains both input and output symbols to produce a FSM showing only either input or output symbols. However, the projection required needs to be able to filter symbols from another FSM.

So, an extension in Java is built, based on the ABB FSM package developed by the Distributed System and Software Engineering Centre of Monash University. The ABB FSM package supports the creation, manipulation and automata operations on FSMs. This package uses the same text file format as the AT&T package, for the creation of FSM. Therefore, the text files created during the data preparation can be reused. In the end, the 'AdapterGeneration' package is implemented as a Java library in order to realize the Specify, Generate and Verify steps in *Adapter Automation*. This package provides functionalities for

- creating ken,
- retrieving FSM of gates
- verifying correctness
- generating 1:2 adapter
- verifying conformance of bindings or mappings

2 classes were created; class 'Ken' and 'Adapter'. Method for correctness checking is called during the instantiation of object from these classes. The main difference between these 2 classes is that provided FSMs and required FSMs are passed into a Ken object as 2 arrays that are non-fixed in size whereas for the Adapter object, the 2 arrays can only store up to 2 FSMs. That is because the algorithm (Schmidt and Reussner, 2002b) used for adapter generation adapts only 2 gates at one time. In other word, to create a 1:3 adaptation, two 1:2 adapters need to be generated.

Provided and required FSM are passed in as parameters during instantiation for correctness checking. This is to ensure that correctness is ensured before any binding or mapping happens. These protocol are checked for their correctness based on the type of gate (provided or required), using the rules defined in the previous Section 5.2.1. The significance of checking the correctness at this early stage will be clearly explained in the *Discussion* Section.

Due to the time constraints and a significant amount of time spent on the preparation of the data, the implementation for the generation of the 2:1 adapter and 1:1 adapter are not complete. Therefore, testing was only done on cases for generation of a 1:2 adapter.

### 6.3 Example from case study

This section presents the example of the DVD component from the DVD\_VCR case study. This example demonstrates the *Adapter Automation* on the DVD component using the ABB package and the AdapterGeneration package.

#### 6.3.1 Step 1: Specify

As explained in the previous chapter, the designer specifies the abstract behaviour and the protocol of the gates in this step and then determine the incompatible connection.

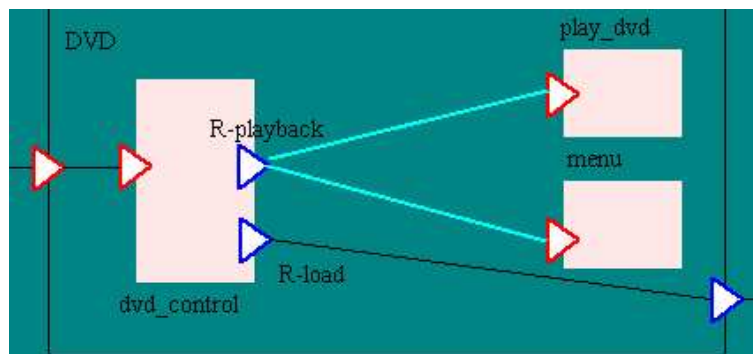


Figure 6.3: DVD ken

Figure 6.3 shows the architecture design of the DVD ken in the DVD\_VCR system. This ken is composed of 3 subkens; *dvd\_control*, *play\_dvd* and *menu*. The *dvd\_control* ken is specified to have one provided gate and 2 required gates, namely the *R-playback* and *R-load* whereas both the *play\_dvd* and *menu* kens have only one provided gate.

First of all, the abstract behaviour and the protocol of the gates for these 3 kens were specified by FSMs. FSMs were created either from the specified text file or by performing automata operations on the FSM using the ABB package. Once the FSMs were available, the object for these 3 kens were instantiated using the AdapterGeneration package. The

correctness of these kens were checked automatically during instantiation.

In this example, we concentrate on the  $R\_playback$  required gate. The  $R\_playback$  required the services of  $play\_dvd$  and  $menu$ , and specified as:

$$R\_playback := play\_dvd \times menu$$

These 2 services were provided by 2 separate kens. So, the two connections (indicated with blue) shown in the Figure 6.3 were not compatible since

$$R\_playback \not\subseteq P\_play\_dvd$$

$$R\_playback \not\subseteq P\_load$$

Therefore, an adapter needed to be introduced to merge the services provided by the two gates into one.

### 6.3.2 Step 2: Generate

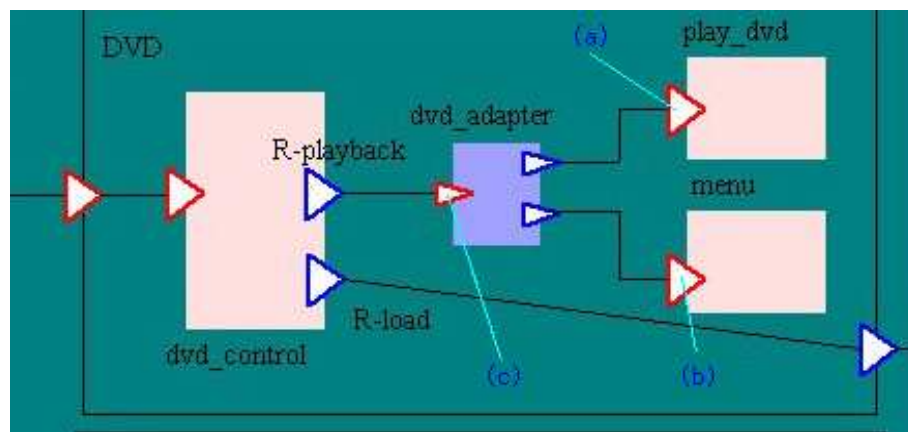
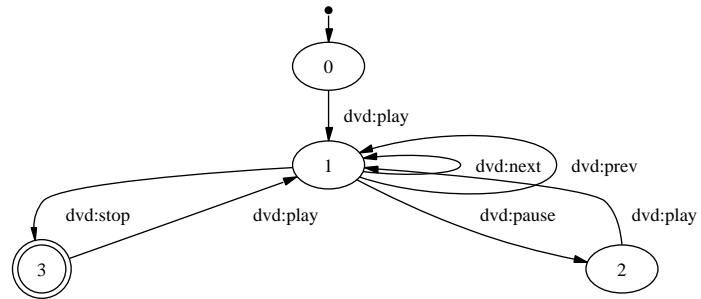
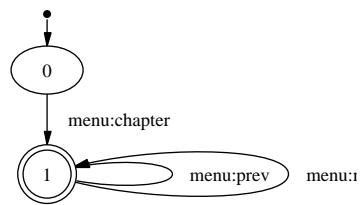


Figure 6.4: A 1:2 adapter is inserted

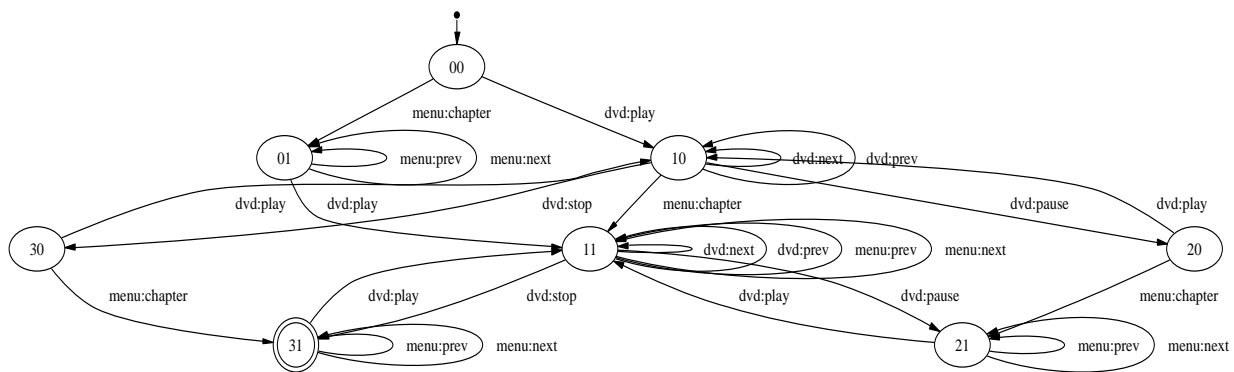
Since the  $R\_playback$  gate required services from 2 kens, a 1:2 adapter was used. Figure 6.4 shows the architecture design with the inclusion of the adapter.



(a) play\_dvd FSM



(b) menu FSM



(c) Provided FSM of adapter

Figure 6.5: FSM in (c) shows all the possible interleavings of FSM in (a) and (b)

The adapter was generated using the function in the AdapterGeneration package. The provided FSMs of the *play\_dvd* and *menu* (annotated with (a) and (b) respectively in Figure 6.4) were passed into that function. Shuffle-product operation was performed on these 2 FSMs and the resulting FSM was returned as the FSM for the abstract behaviour of the adapter. The resulting FSM shows all the possible interleavings of provided FSMs *play\_dvd* and *menu*. This scenario is clearly shown in Figure 6.5.

Since no further restrictions or operations were required, the FSMs for the provided gate and the abstract behaviour of the adapter were the same.

A scenario where further restriction is required is described in the *Discussion* Section.

### 6.3.3 Step 3: Verify

Once the adapter has been finalized, compatibility verification of the adapter that was generated earlier (refer to adapter in Figure 6.4) is executed. As explained in previous chapter, there are 2 parts to the verification; verifying the correctness and the conformance. Verification of correctness and verification of conformance are done using the functions from the AdapterGeneration package.

#### Verify correctness:

The adapter generated has one provided gate and two required gates (see Figure 6.4). All three gates were verified for their correctness. The result returned indicated that all three gates were correct and therefore the adapter was correct. This can be proven mathematically as below:

$$\begin{array}{ll}
 R1 := \textit{play\_dvd} & (\textit{first required gate}) \\
 R2 := \textit{menu} & (\textit{second required gate}) \\
 A := \textit{play\_dvd} \times \textit{menu} & (\textit{abstract behaviour}) \\
 P := \textit{play\_dvd} \times \textit{menu} & (\textit{provided gate})
 \end{array}$$

$$\begin{array}{ll}
 R1' := (A) \setminus R1 & (\textit{projection of R1 over A}) \\
 := (\textit{play\_dvd} \times \textit{menu}) \setminus \textit{play\_dvd} \\
 := \textit{play\_dvd}
 \end{array}$$

$$\begin{aligned}
R2' &:= (A) \setminus R2 && \text{(projection of } R2 \text{ over } A) \\
&:= (\text{play\_dvd} \times \text{menu}) \setminus \text{menu} \\
&:= \text{menu}
\end{aligned}$$

$$\begin{aligned}
P' &:= (A) \setminus P && \text{(projection of } P \text{ over } A) \\
&:= (\text{play\_dvd} \times \text{menu}) \setminus \text{play\_dvd} \times \text{menu} \\
&:= \text{play\_dvd} \times \text{menu}
\end{aligned}$$

$$\begin{aligned}
L(R1') \subseteq L(R1) &\Rightarrow (\text{minimize}(R1') == \text{minimize}(R1' \cap R1)) \ \&\& \\
(\text{minimize}(R1') == \text{minimize}(R1' \cap R1)) &\Rightarrow \text{true} \\
\Rightarrow L(R1') \subseteq L(R1) &\text{ is true} && \text{(gate } R1 \text{ is correct)}
\end{aligned}$$

$$\begin{aligned}
L(R2') \subseteq L(R2) &\Rightarrow (\text{minimize}(R2') == \text{minimize}(R2' \cap R2)) \ \&\& \\
(\text{minimize}(R2') == \text{minimize}(R2' \cap R2)) &\Rightarrow \text{true} \\
\Rightarrow L(R2') \subseteq L(R2) &\text{ is true} && \text{(gate } R2 \text{ is correct)}
\end{aligned}$$

$$\begin{aligned}
L(P) \subseteq L(P') &\Rightarrow (\text{minimize}(P) == \text{minimize}(P \cap P')) \ \&\& \\
(\text{minimize}(P) == \text{minimize}(P \cap P')) &\Rightarrow \text{true} \\
\Rightarrow L(P) \subseteq L(P') &\text{ is true} && \text{(gate } P \text{ is correct)}
\end{aligned}$$

*Adapter is correct since*

$$(L(R1') \subseteq L(R1)) \ \&\& \ (L(R2') \subseteq L(R2)) \ \&\& \ (L(P) \subseteq L(P'))$$

*is true.*

\*Note the different rules being applied to the required gate ( $R1$ ,  $R2$ ) and the provided gate ( $P$ ). Refer to Section 5.2.1 for the definition of the rules.

### **Verify conformance:**

After the correctness verification, the connections of the adapter were verified for their conformance. Refer to Figure 6.4 to see where the connections are. In this case, the only type of connections is the **binding**. Therefore only the rule for checking the conformance of binding is used (refer to Section 5.2.3 for the definition of the rule). The result returned indicated that all the bindings are valid. Therefore, the adapter conformed. This can be proven mathematically as below and refer to Figure 6.4 in order to understand the working:

\* $R1$ ,  $R2$  and  $P$  are defined previously in the verify correctness part.

$p1 := play\_dvd$	<i>(provided gate for ken play_dvd)</i>
$p2 := menu$	<i>(provided gate for ken menu)</i>
$r := play\_dvd \times menu$	<i>(R_playback required gate)</i>

$RP1 := binding\ for\ (R1\ to\ p1)$

$RP2 := binding\ for\ (R2\ to\ p2)$

$RP3 := binding\ for\ (r\ to\ P)$

Adapter conforms if the bindings of  $RP1$ ,  $RP2$  and  $RP3$  are valid.

$L(R1) \subseteq L(p1) \Rightarrow (minimize(R1) == minimize(R1 \cap p1)) \ \&\&$   
 $(minimize(R1) == minimize(R1 \cap p1)) \Rightarrow true$   
 $\Rightarrow L(R1) \subseteq L(p1)\ is\ true$  *(binding RP1 conforms)*

$L(R2) \subseteq L(p2) \Rightarrow (minimize(R2) == minimize(R2 \cap p2)) \ \&\&$   
 $(minimize(R2) == minimize(R2 \cap p2)) \Rightarrow true$   
 $\Rightarrow L(R2) \subseteq L(p2)\ is\ true$  *(binding RP2 conforms)*

$L(r) \subseteq L(P) \Rightarrow (minimize(r) == minimize(r \cap P)) \ \&\&$   
 $(minimize(r) == minimize(r \cap P)) \Rightarrow true$   
 $\Rightarrow L(r) \subseteq L(P)\ is\ true$  *(binding RP3 conforms)*

*Adapter conforms since*

$(L(R1) \subseteq L(p1) \ \&\& \ (L(R2) \subseteq L(p2) \ \&\& \ (L(r) \subseteq L(P)))$

*is true.*

Since the adapter was correct and conformed, it was concluded that the adapter generated was compatible.

## 6.4 Discussion

Adapter Automation that is devised in this research is closely related to the component adaptation methodology presented in by Bracciali et al. (n.d.). As explained in Section 4.1, their methodology comprises of 4 main aspects; *component interfaces*, *adapter specification*, *adapter derivation* and *adapter properties* as opposed to this Adapter Automation methodology that has only 3 steps. Instead of specifying the behaviour of the adapter and then deriving the adapter, in Adapter Automation, adapter is generated first automatically and then further restriction may applied to the behaviour of the adapter as seen fit by the designer (user-driven). This way allows more flexibility in the designer's control over the adapter generation process. The other difference between the two methods lies in the mechanism for protocol specification. Bracciali et al.'s (n.d.) approach is based on using process algebra in specification of interfaces and adapters as well as in the derivation and verification of the adapter. On the other hand, Adapter Automation concentrates on using Finite State Machines in all the specification and verification of protocol, based on architectural context. FSM is chosen over process algebra due to the simplicity of FSM that encourages the verification of compatibility.

FSMs of the provided and required gates are passed in as parameters during the instantiation of ken for correctness checking (in the Specify step). An important advantage of checking for correctness at this stage is that, the occurrence of an error at this stage indicates that design change is required rather than adapter generation. If correctness checking is not done in this stage, incompatibility that occurs later on due to that error would be hard to be pinpointed.

Another advantage in detecting error at an early stage is to reduce the effect of the error on other kens. If a design error is only detected at a later stage, modification to the design of that ken may raise error for other kens due to the dependency relationships of the kens, i.e. ken either uses other kens or being used by other kens.

Instead of only detecting whether an error has occurred, the checking reveals where the design error is and may even show the correct design as result (see Figure 2 and Figure 3 of the Appendix). This is made possible due to the modeling of protocol specification with FSM.

During the *Generate* step, further restriction or operations may be required. The following describes a scenario in which further restriction is needed.

Let  $vcr$  and  $dvd$  be 2 provided gates providing the following services

$$\begin{aligned} vcr &:= play\_vcr + record\_vcr \\ dvd &:= play\_dvd \times menu \end{aligned}$$

and required gate  $R\_X$  requires the following services

$$R\_X := record\_vcr \times dvd$$

So, adapter is generated to adapt  $vcr$  and  $dvd$ . The abstract behaviour of the adapter is

$$\begin{aligned} adapter &:= vcr \times dvd \\ &:= (play\_vcr + record\_vcr) \times dvd \end{aligned}$$

Restriction is required to remove  $play\_vcr$ . So, the provided gate for the adapter becomes

$$\begin{aligned} P\_adapter &:= adapter \setminus (dvd \times (record\_vcr)) \\ &:= dvd \times (record\_vcr) \end{aligned}$$

and when connected to  $R\_X$ , the binding is valid since  $L(R\_X) \subseteq L(P\_adapter)$  is satisfied.

Even if restriction is not performed,

$$P\_adapter := adapter$$

the binding between  $R\_X$  and  $P\_adapter$  still conforms since

$$\begin{aligned} L(R\_X) &\subseteq L(P\_adapter) \\ record\_vcr \times dvd &\subseteq (play\_vcr + record\_vcr) \times dvd \end{aligned}$$

is still satisfied. However, the extra transitions in the interleavings that are not required will only increase the complexity of analysis of the FSM and as a result *may* cause the compatibility verification in the *Verify* step to become slower. This effect is not shown in this example since the DVD\_VCR system is small, but may be shown for a very large and complex system.

If an adapter is found incompatible during the compatibility verification, further automata operations may be performed to resolve the incompatibilities. However, not all incompatibilities can be removed. An obvious one would be when the provided FSM does not have a symbol required by the required gate. Adapter generation will not solve this since the adapter only manipulates the existing FSM to resolve incompatibility, it *cannot* add a new symbol to the provided FSM.

## Chapter 7

# Conclusion

This research project is motivated by the need to introduce a systematic way of generating adapters to resolve the incompatibilities that arise through architecture description and protocol specification using FSM. The incompatibilities that we focused on are the behavioural incompatibilities in the context of architecture reuse, i.e. reusing of existing components and interconnecting these components. So, this chapter present a review on how the objective of this research has been achieved and the limitations.

This thesis has presented the concept of a systematic method of generating adapter in order to overcome the behavioural incompatibilities that arise. The concept of **Adapter Automation** is introduced ot refer to the method. It is proposed for this method to be consisted of the 3 main steps; namely the *Specify*, *Generate* and *Verify*. Specifying the abstract behaviour and interface protocol of components, verify the correctness of these interfaces and distinguish between compatible and incompatible connections in the first step. Generating adapter to overcome incompatibility in step 2. Verifying the compatibility of the adapter generated with its environment in step 3. In order to realize the Adapter Automation, an extension to the existing ABB FSA package is built as Java library (AdapterGeneration package).

To show the reuse of component, interconnection between components and to present extra specification of a component (protocol specification in this case), architecture description is used. The architecture concept of ken, gate, binding and mapping is adopted. The usage of architecture description has resulted in a better understanding of the design and interoperability of a system and thus enable designer to detect some incompatibility based on the architecture design diagram. This is experienced when designing the architecture of the DVD\_VCR player, e.g. when 2 connection lines are coming out from a required gate.

In order to allow for automatic adapter generation, interface protocol specification have to be formalized as shown in Schmidt and Reussner's (2002b) and Bracciali et al.'s (n.d.) papers. Therefore, an automata-based approach has been adopted by modelling protocol specification with FSM. In other word, by formalizing protocol specification, incompatibility can be automatically removed by the automatic generation of adapter. However, not all incompatibilities can be removed automatically. Some may be removed through automated checking and user-driven functions from the package. Based on the architecture design, it is found that 3 types of adapters are mostly need; 1:2 adapter, 2:1 adapter and the 1:1 adapter. However, the adapter generation is only partially implemented in the package due to the time constraints. So far, only adapter generation of a 1:2 adapter has been implemented.

It has been demonstrated in this thesis that verification of correctness and conformance plays a very significant role in Adapter Automation (in the Specify and Verify steps). The main purpose of the verification is to ensure compatibility of the adapter with its environment. However, other benefits are also shown in this thesis. It helps in the user-driven generation of adapter. Apart from that, it also shows that if error is detected in the Specify step, removal of incompatibilities requires design modification rather than adapter generation. Another unexpected benefit is that the verification (in the Specify step) reveals where the design error is and may suggest the correct design (by comparing the graphical notation of FSMs).

In conclusion, the Adapter Automation method, which adopts the concept of architecture description and protocol specification through FSM, is able to remove some incompatibilities automatically through automatic generation of adapter. Some other incompatibilities may be removed through further user-driven functions and some are not removable through by generating adapter. A lot of effort still has to be put into extending the AdapterGeneration package, especially for the other 2 common adapters and to make it more interactive with the user.

# References

- Allen, R. and Garlan, D. (1997). A formal basis for architectural connection, *ACM Transactions on Software Engineering and Methodology (TOSEM)* **6**(3): 213–249.
- Bracciali, A., Brogi, A. and Canal, C. (n.d.). Systematic component adaptation.  
\*citeseer.nj.nec.com/552491.html
- Brand, D. and Zafropulo, P. (1983). On communicating finite state machines, *Journal of the ACM* **30**: 323–342.
- de Alfaro, L. and Henzinger, T. A. (2001). Interface automata, *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ACM Press, pp. 109–120.
- Garlan, D., Allen, R. and Ockerbloom, J. (1995). Architectural mismatch or why it’s hard to build systems out of existing parts, *Proceedings of the 17th international conference on Software engineering*, ACM Press, pp. 179–185.
- Heineman, G. T. (1999). Adaptation of software components.  
\*citeseer.nj.nec.com/heineman99adaptation.html
- Jazayeri, M., Ran, A. and der Linden, F. V. (2000). *Software Architecture for Product Families*, Addison Wesley.
- Marangozov, V., Bellissard, L., Vion-Dury, J.-Y. and Riveill, M. (1997). Connectors: a key feature for building distributed component-based architectures.  
\*citeseer.nj.nec.com/marangozov97connectors.html
- Passerone, R., de Alfaro, L., Henzinger, T. A. and Sangiovanni-Vincentelli, A. L. (2002). Convertibility verification and converter synthesis: two faces of the same coin, *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, ACM Press, pp. 132–139.
- Schmidt, H. W. and Reussner, R. H. (2002a). Generating adapters for concurrent component protocol synchronisation, *Formal Methods of Open Object-based Distributed Systems(FMOODS) V*, Kluwer Academic Publication, pp. 213–229.

- Schmidt, H. W. and Reussner, R. H. (2002b). Parameterised contracts and adapter synthesis, *Proc. 5th Intl Component-Based Software Engineering Workshop(CBSE5) of the ICSE Conference*.
- Shaw, M. (1995). Architectural issues in software reuse: it's not just the functionality, it's the packaging, *Proceedings of the 1995 Symposium on Software reusability*, ACM Press, pp. 3–6.
- Vallecillo, A., Hernandez, J. and Troya, J. (2000). Component interoperability. [\\*citeseer.nj.nec.com/vallecillo00component.html](http://citeseer.nj.nec.com/vallecillo00component.html)
- Yellin, D. M. and Strom, R. E. (1995). Interfaces, protocols, and the semi-automatic construction of software adaptors, *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, ACM Press, pp. 176–190.

# Diagrams

**Components of the DVD\_VCR player specified by the following regular expressions:**

See next page for the architecture design diagram.

- + *represents or*
- × *represents shuffle-product*
- *represents concatenate*

Regular expressions for the 3 main components; VCR, DVD and VCD:

$$\begin{aligned} vcr\_control &:= tapeloder \bullet (play\_vcr + record\_vcr) \\ dvd\_control &:= discloader \bullet (play\_dvd \times menu) \\ vcd\_control &:= discloader \bullet (play\_vcd) \end{aligned}$$

Regular expressions for the 3 required gates of the 'main\_control':

$$\begin{aligned} R\_play\_vcr\_only &:= tapeloder \bullet play\_vcr \\ (R\_record\_vcr \times dvd) &:= (tapeloder \bullet record\_vcr) \times (discloader.menu \times play\_dvd) \\ R\_vcd &:= discloader \bullet (play\_vcd) \end{aligned}$$

Regular expression for the 'main\_control':

$$main\_control := power \bullet (play\_vcr\_only + (record\_vcr \times dvd) + play\_vcd)$$

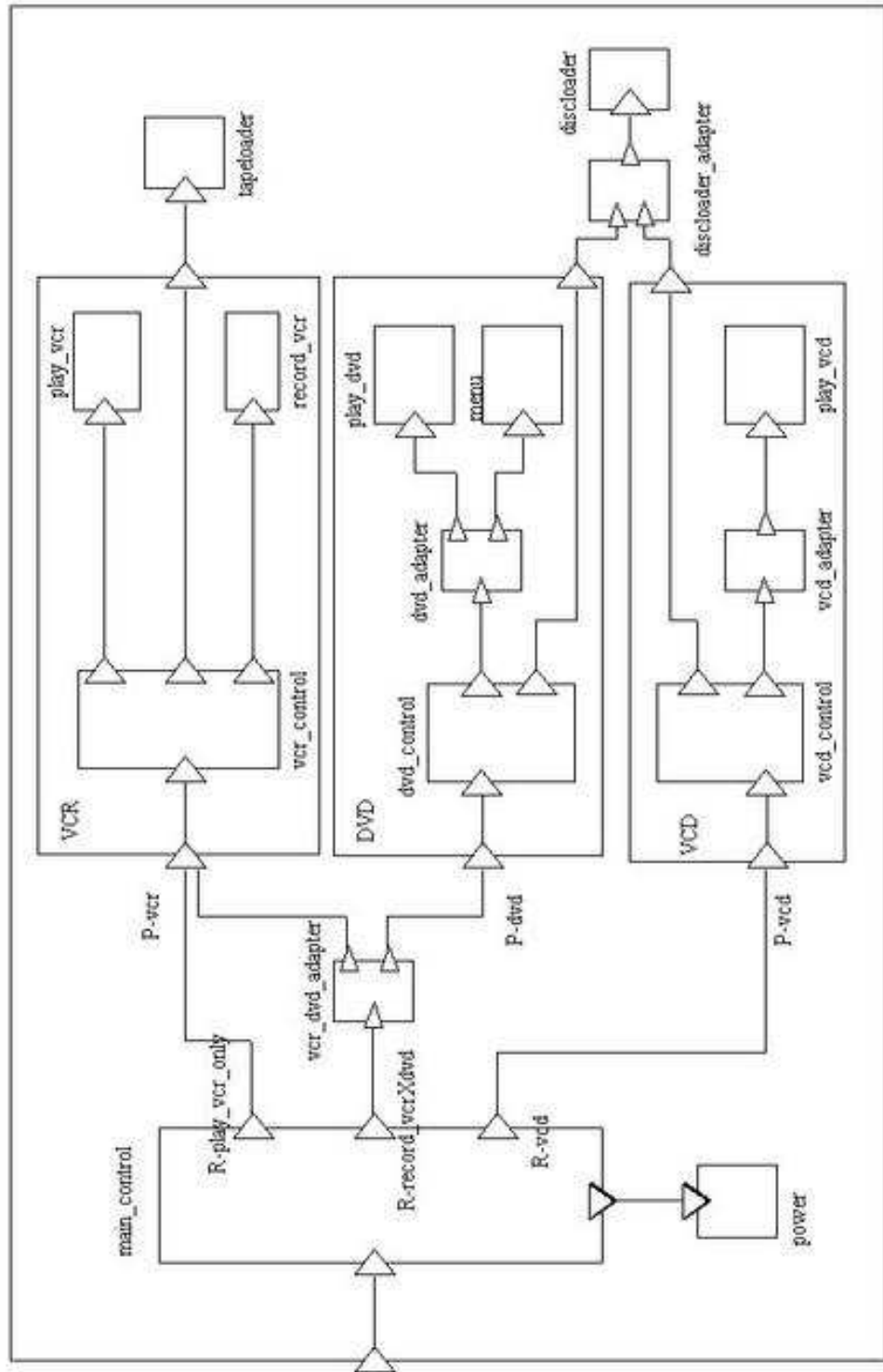
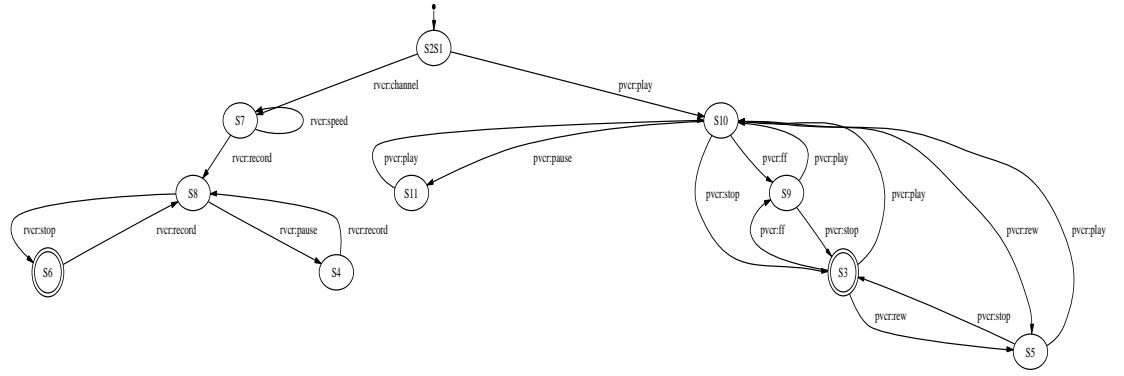
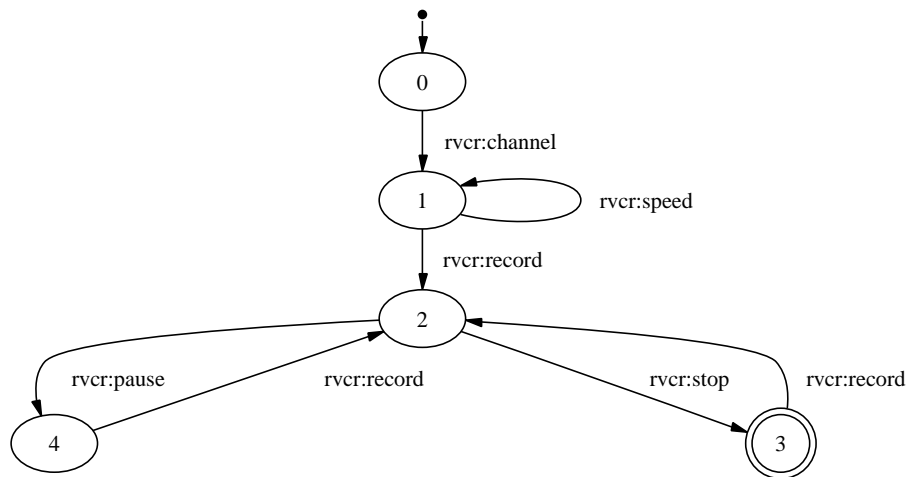


Figure 1: Architecture design of a DVD\_VCR player. See previous page for the regular expressions of the main components.

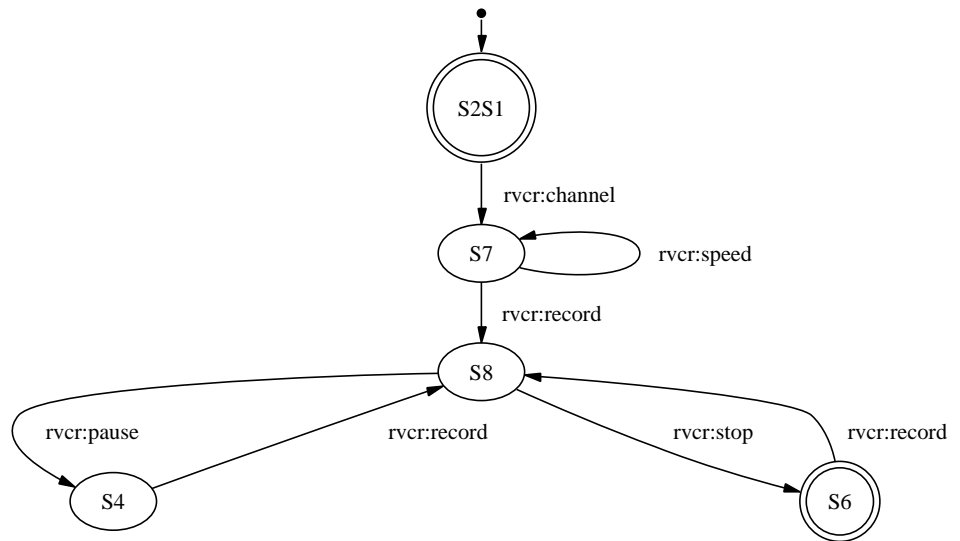


(a) Abstract behaviour of `vcr_control` (`play_vcr` + `record_vcr`): computed using package

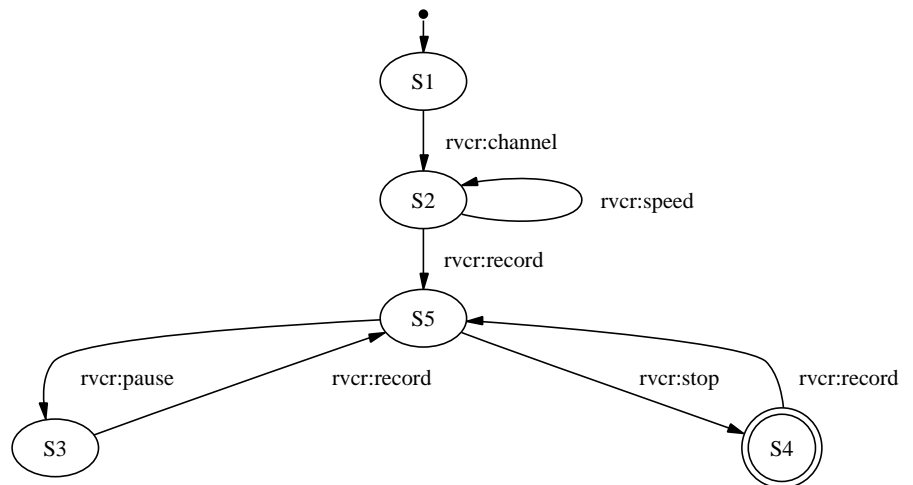


(b) Required FSM `record_vcr` for ken `vcr_control` (`record_vcr`): specified by user in text file

Figure 2: FSMs **before** correctness checking of `record_vcr`



(a) Projection of record\_vcr over abstract of vcr\_control



(b) Intersection of the result from projection and record\_vcr

Figure 3: FSMs returned **after** the correctness checking. Error is revealed to be located in the Start state. This diagram suggests to the designer that, the start state is also the initial state, may be the correct design.