A Formal Verification Approach of Conversations in Composite Web Services

Melissa Kova

A Thesis

in

the Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science (Quality Systems Engineering) at Concordia University Montreal, Quebec, Canada

August 2009

Your file Votre référence
ISBN: 978-0-494-63062-4
Our file Notre référence
ISBN: 978-0-494-63062-4

ABSTRACT

A Formal Verification Approach of Conversations in Composite Web Services

Melissa Kova

Web service composition is nowadays a very focused-on topic of research by academic and industrial research groups. This thesis discusses the design and verification of behaviors of composite web services. To model composite web services, two behaviors are proposed, namely control and operational. The operational behavior shows the business logic of the process functionality for a composite web service. The control behavior shows the constraints that the operational behavior should satisfy and specifies the states that this behavior should be in. The idea behind this separation is to promote the design, verification and reusability of web services in composite settings. To guarantee their compatibility, these two behaviors communicate and synchronize through conversation messages. State charts are used to model composite web services and symbolic model checking with NuSMV model checker is used to verify their conversations. The properties to be verified are expressed in two logics: Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). A Java-based translation procedure from the design model to SMV program used by NuSMV has been developed and tested in two case studies.

ACKNOWLEDGMENTS

A warm appreciation is extended to all the faculty and staff at the Concordia Institute for Information Systems Engineering at Concordia University, whose suggestions and encouragement were invaluable in my courses throughout my first year at the university.

I would also like to thank Dr. Jamal Bentahar for his guidance and his constructive and helpful remarks throughout the duration of this thesis.

Last but not least, I would like to thank my parents for their continual support throughout my years of study. Without their help, this degree would not have been possible. I dedicate this work to all my family and friends.

Table of contents

List of Figures

List of Tables

List of Acronyms

ATM:	Automated Teller Machine
BPEL:	Business Process Execution Language
BPEL4WS:	Business Process Execution Language for Web Services
CTL:	Computational Tree Logic
HTTP:	Hypertext Transfer Protocol
LTL:	Linear Temporal Logic
OBDDS:	Ordered Binary Decision Diagrams
OWL-S:	Semantic Markup for Web Services
SMTP:	Simple Mail Transfer Protocol
SMV:	Symbolic Model Verifier
SOAP:	Simple Object Access Protocol
UDDI:	Universal Description Discovery and Integration
URI:	Uniform Resource Identifier
WSCI:	Web Service Choreography Interface
WSD:	Web Service Description
WSDL:	Web Service Description Language
WSFL:	Web Service Flow Language
W3C:	World Wide Web Consortium
XLANG:	XML-based language
XML:	Extensible Markup Language

Chapter 1: Introduction

In this chapter, we introduce the context of our research and explain what initiated our interest into the design and implementation of composite web services. We also present the research questions that we considered, a general description of our objectives and contributions and an overview of the structure of the thesis.

1.1 Context of Research

The World Wide Web Consortium (W3C) organization, which establishes the standards for web services, defines them as follows: "A web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols. Web services are characterized by their great interoperability and extensibility, as well as their machine-processable descriptions thanks to the use of XML. They can be combined in a loosely coupled way in order to achieve complex operations. Programs providing simple services can interact with each other in order to deliver sophisticated added-value services" [21].

Web services are emerging nowadays and the best quality of the conversation among composite web services should be assured. Web services have become the primary infrastructure for varied interconnection of business processes, systems and products so it is crucial to have a reliable delivery of messages. The participants should be sure of the completion of message exchanges to be able to solve the different business problems. When this completion is ensured, we can define the related web service as secure, interoperable and transactional.

Composition in web services brings value-added benefit and flexibility. In fact, composition is an important aspect in web services and should be supported by the architecture that contains the protocols and interfaces for reliable message exchanging in order to provide the functions that customer, software vendors and industries need.

In this thesis, our purpose is to formalize composite web services and then apply model checking to verify the conformity of the model we propose. The idea of modeling and studying the web services under two different behaviors: control and operational behaviors was previously studied in [27] and [39]. In these two publications, the control behavior illustrates the business logic that underpins the functioning of an isolated web service, and the operational behavior regulates the execution progress of this control behavior by stating the actions to carry out and the constraints to put on this progress. However, the composition and verification aspects were not investigated. The technique we are using in this thesis combines this idea of separating control and operational behaviors with an additional approach. So, in this thesis, our aim is to develop an efficient and easy to use verification model for composite web services.

1.2 Motivations

To achieve a highest quality in conversations among web services, some verification should be done on these conversations. Our first motivation is to have the fastest and most reliable web access service. Therefore, we present a general framework accessible by the users to check the correctness in the transfer of messages. Our second motivation is to assure that this framework can show if there is a problem or not. So we apply a model checking technique to verify the correctness of the properties that assure the good quality in the communication.

Because a reliable message delivery standard will improve the effectiveness of other web services standards, like security, transactions and business processes our final motivation will be to assure this reliability by proving the efficiency of the proposed verification model through simulation of an example using NuSMV model checker [7] and the program code we developed in JAVA.

1.3 Research Questions

The research questions that are considered in this thesis are:

- How can we design composite web services to ensure good quality in the mechanism of message exchanging?
- How state charts can be used to verify the reliability of the communication between web services?
- What kind of composition will we consider in the design of our framework?
- How model checking can be considered a good verification technique for composite web services?

1.4 Contributions

Our main objective is to verify the conversations among composite web services to be able to give the users a best quality.

The contributions of this thesis can be summarized as follows:

- An approach for modeling composite web services based on two behaviors: control and operational. These two behaviors are linked together to check the synchronization between the conversations in composite scenarios. We use state charts enhanced with additional syntax to facilitate the mapping process between the two behaviors to model composite web services.
- A formal and automatic verification approach of the mapping procedure using symbolic model checking technique. The implementation is done using a Java-based translation procedure and NuSMV model checker [7].

1.5 Thesis Overview

The thesis is divided into 6 chapters. Chapter 1 introduces our work and presents our motivations and contributions. Chapter 2 introduces composite web services and discusses two different types of composition. Chapter 3 presents model checking technique along with two logics for which model checking is used: Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). In Chapter 4, we propose our model and support it by two different use cases: a ticket reservation system and an ATM system. Chapter 5 shows the different implementations we have done to translate state charts to SMV programs that are the inputs of NUSMV model checker. Finally, Chapter 6 concludes our work and presents some direction for future work.

Chapter 2: Composite Web Services

2.1 Overview of Web Services

Web services are modular and self-contained applications that are described, published, located and invoked over a network: the World Wide Web [20]. They are based on open Internet standards like XML (Extensible Markup Language), HTTP (Hypertext Transfer Protocol) and SMTP (Simple Mail Transfer Protocol), and do not rely on a specific operating system, language or environment. In [34], the definition of a web service is given as: "any process that can be integrated into external systems through valid XML documents over Internet protocols".

Web services are based on specifications for data transfer, method invocation and publishing. It is important to emphasize that a web service is a service that should include an interface to communicate with other applications via SOAP (Simple Object Access Protocol). According to the previous definition, a weather forecast on a web page for example is not necessary a web service. It is considered a web service if it communicates with other software components.

Web service must be implemented by a concrete agent (software) that sends and receives messages (See Figure 2.1) [5]. In this figure, we note that the provider entity is the person or organization responsible for providing the agent implementing the service. However, the requester (consumer) entity will use a requester agent to communicate with the provider's agent by exchanging messages. The messages exchanged between the provider and requester is documented in a Web Service Description document (WSD). This description is expressed in WSDL (Web Service Description Language), which is often used in combination with SOAP and XML Schema. A client program connecting to a web service can determine what functions are available on the server by reading the WSDL file. The data types used are embedded in this file and the client will use SOAP to call one of the functions in the file.

In fact, the WSD represents a contract, it specifies the message formats, transport protocols and location. One more network location where the provider can be invoked will be specified.

The main elements used in the definition of network services in a WSDL document are:

- Types: a container for data type definitions using some type system to describe the messages exchanged.
- Message: an abstract description of the data being exchanged. A message consists of logical parts, each of which is associated with a definition within some type system.
- Operation: an abstract description of an action supported by the service. Each operation refers to an input message and output messages.
- Port Type: abstract collections of operations supported by one or more endpoints.
- Binding: a concrete protocol and data format specification for a particular port type.
- Port: a single endpoint defined as a combination of a binding and a network address.
- Service: a collection of related endpoints/ports.

These elements will not be described in details because they are not our main focus in this work.

"Web services are considerably expanding and being used for many purposes such as integrated enterprise applications, business-to-business collaborations and e-government systems. Web services represent a further evolution in distributed computing technologies. They are a set of standardized technologies that operate on common protocols to facilitate the access to remote services in a standardized, vendor-neutral way. Although these technologies are mature, web services still have to encompass additional features (verification, security, transaction-handling, session-handling, etc.) to facilitate robust, dynamic business services" [18]

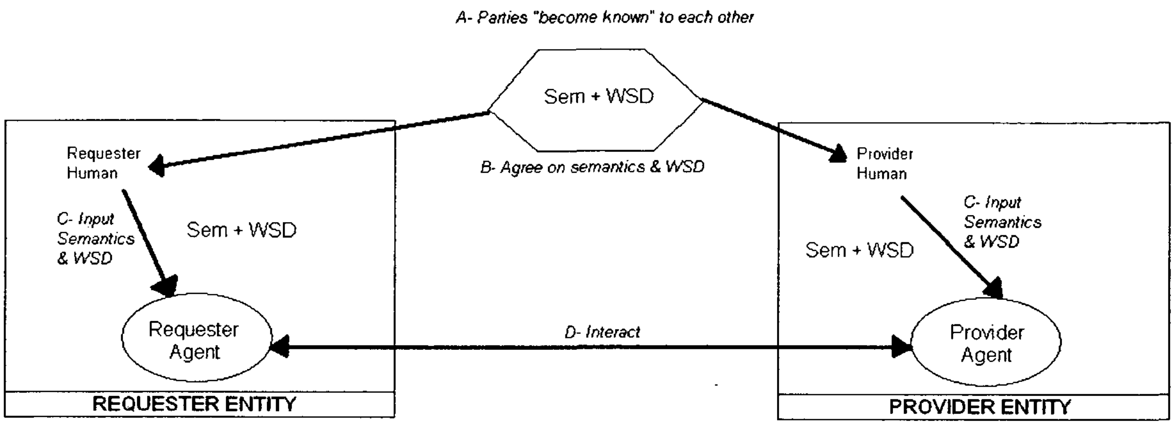


Figure 2.1 The General Process of Engaging a Web Service

2.2 Definition of Composite Web Services

Web services provide the basis for the development and execution of business processes that are distributed over the network and available via standard interfaces and protocols. Service Composition [23] is very promising in web domain. When combining component web services (existing web services), we will have composite web services that can execute a lot of new functionalities. An advantage of composition is the reduction of development time and effort to make new applications.

Web service composition is an active area of research. The earliest languages to define standards for web services composition were IBM's Web Service Flow Language (WSFL) [25] and Microsoft's XLANG (XML-based language) [38]. These two languages were an extension of the WSDL. WSDL is used to describe the syntactic aspects of a web service.

BPEL4WS (Business Process Execution Language for Web Services) is one of the emerging standards for describing the behavior of the services. It is a recent language that merges the graph oriented representation in WSFL and the structural construct based processes of XLANG.

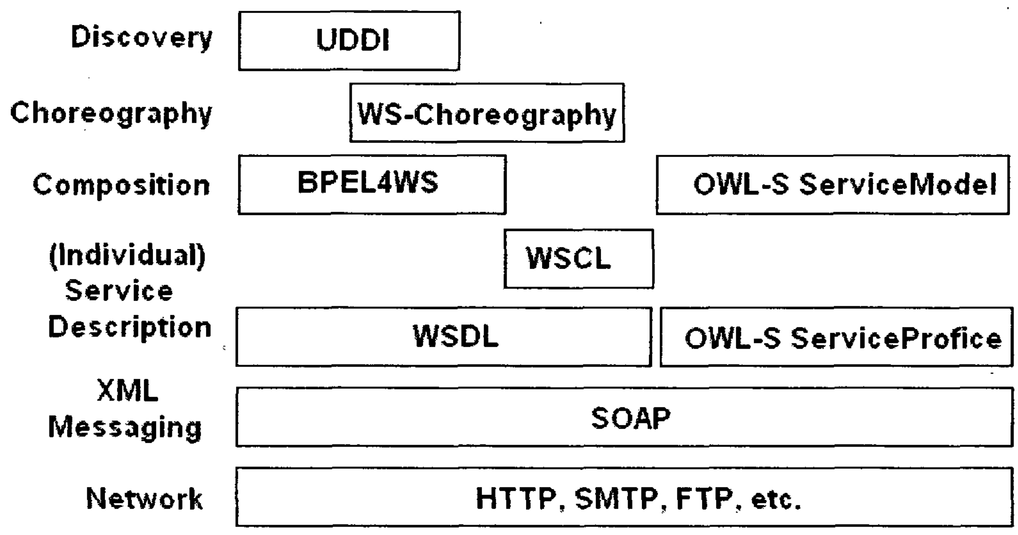


Figure 2.2 Web Service Standards Stack

Figure 2.2 illustrates the Web Service Standards stack [22]. These standards enable a flexibility in combining web services to create more complex ones. UDDI (Universal Description Discovery and Integration) allow manual and automated discovery of web services and helps in the creation of composite web services. BPEL (Business Process Execution Language) is used to coordinate the activities of web services in a procedural language. OWL-S (Semantic Markup for Web Services) language "describes web services in terms of their inputs, outputs, preconditions and effects, and of their process model" [22]. Users and agents should be able to automatically discover, negotiate with, compose, invoke and monitor web services.

There are two main types of composition: static and dynamic composition [4]:

In static composition, the service to which the agents are going to be connected is determined before the execution of the flow. An example of static composition is the information for tourists in a travel service such as a list of places of interests, list of car rentals, etc. The existence of such a service is then known before run-time.

In dynamic composition, some of the services are not known during the design time and they are only known during run-time. For example, if we want to find the lowest price of an air ticket for a particular destination. During run-time, the agent will connect to all the available ticket booking services to be able to choose the lowest price.

To resume, we use static composition when the nature of the process to be composed is fixed and when the business partners and services are slowly changing. However, we use dynamic composition if the process has mostly undefined functions to perform and it has to adapt to changes in the environment dynamically.

Composite services could be mandatory or optional. A composite service is mandatory when all the component services participate in the execution process. However, an optional composite service does not necessarily involve all the component services. Some services do not participate in the execution because of non availability or because of substitution.

We discussed several types of compositions. There exist two important approaches to composition [2] [27]: orchestration and choreography [16] [17] [28]. On the one hand, choreography specification identifies the set of allowable conversations for a composite web service. An orchestration, on the other hand, is an executable specification that identifies the steps of execution for the peers. They will be explained in the following section.

2.3 Types of Composition

Each web service performs one distinct functionality. When combining these individual components we can make an entire application work. There exist two different ways of combination: orchestration and choreography. The main difference between them is that orchestration has a central controller while choreography does not. Section 2.3.1 explains in details the orchestration method and one language related to it: BPEL4WS. Section 2.3.2 defines the choreography method and the WSCI language.

2.3.1 Orchestration

Orchestration can be basically defined as an orchestra where the leader directs all the musicians on what to do. Therefore, the musicians are synchronized by following the direction of one person. In practice when orchestration is in place, a central system says to some remote systems what to do. Figure 2.3 shows how the messages are transferred between the different web services. We can see that the process in orchestration is always controlled from one of the business parties. This central process can be another web service. It should be aware of the different operations used in the process as well as the order the other web services are invoked in. The other web services usually do not know that they are involved in a composition scenario. They do not need to know that. The interaction is done at the message level [29] [30] [31].

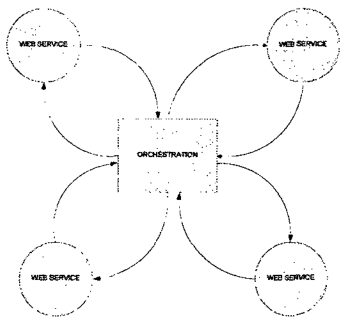


Figure 2.3 Orchestration Schema

One language used in orchestration process is BPEL4WS (Business Process Execution Language for Web Services). "The BPEL4WS provides an XML-based grammar for describing the control logic required to coordinate web services participating in a process flow and is layered on top of WSDL, with BPEL4WS defining how the WSDL operations should be sequenced. BPEL4WS provides support for both abstract business protocols and executable business processes. A BPEL4WS business protocol specifies the public message exchanges between parties. Business protocols are not executable and do not convey the internal details of a process flow, similar to WSCI. An executable process models the behavior of participants in a specific business interaction, essentially modeling a private workflow. Executable processes provide the orchestration support described earlier, while the business protocols focus more on web services choreography" [30].

In BPEL4WS, the activities of a process are structured; they could be sequential and parallel. BPEL4WS also supports conditional looping and dynamic branching. There are two important elements in BPEL: the variables and partners. Variables refer to the data exchanged in the message flow. "When a BPEL4WS process receives a message, the appropriate variable is populated so that subsequent requests can access the data" [30]. Whereas, partners are all the different parties that participate in the process.

The typical scenario of orchestration consists of receiving a message into a BPEL executable process. Then, the process will invoke the concerned web services to be able to respond back to the requestor at the end.

2.3.2 Choreography

Simply, choreography can be compared to a dancing stage where every dancer knows exactly what to do, and looks to all the other dancers involved in the process, to synchronize his steps. A single remote system knows what to do and also what other systems to call after he ends his processing. Choreography is more collaborative in nature. Therefore, each party involved in the process should describe the part they play in the interaction [29] [30].

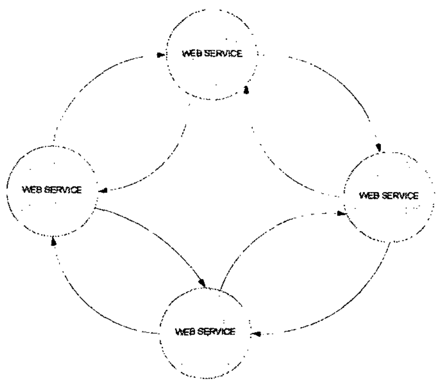


Figure 2.4 Choreography Schema

Choreography does not rely on a central coordinator like orchestration. All the web services that are involved in the composition scenario should know exactly with whom to interact an when to execute their operations. Choreography relies on the exchange of messages in public business processes. Therefore, as said, all participants in the choreography need to be aware of the business process, operations to execute, messages to exchange, and the timing of message exchanges. Figure 2.4 shows the basic interaction of the exchanging of messages that is done in choreography.

The choreography language is WSCI (Web Service Choreography Interface). The Web Service Choreography Interface (WSCI) is an XML-based interface description language that describes the flow of messages exchanged by a web service participating in choreographed interactions with other services. WSCI only describes the observable behavior (messages exchanged) between the different web services. In choreography we will have a set of WSCI interfaces, one for each partner in the interaction. "WSCI can be viewed as a layer on top of the existing web services stack. Each action in WSCI represents a unit of work, which typically would map to a specific WSDL operation. WSCI defines an <action> tag for specifying a basic request or response message. Each activity specifies the WSDL operation involved and the role being played by the participant. External services can then be invoked through the <call> tag. A wide variety of structured activities are supported, including sequential and parallel processing and condition looping. WSCI also introduces an <all> activity, used to indicate that the specific actions have to be performed, but not in any particular order" [30].

Choreography won't be used in this work but it was introduced because it could be interesting to work on it in future works.

2.3.3 Orchestration vs. Choreography

The main difference between orchestration and choreography is that orchestration is controlled by a single party whereas in choreography no one controls the conversation. In orchestration the other web services do not know about the process. Only the central controller is aware of the flow of the process. However, in choreography, all the web services are aware of the process and of whom to interact with because they exchange messages between themselves.

In fact, we can say that orchestration is a controlled and coordinated way of utilizing the services of all the participating web services whereas choreography is just a collaborative effort of utilizing the services of the participating web services.

Also, regarding the fault handling issue, it is easier in orchestration as the execution is controlled, which is not the case with choreography. Web services can be easily and transparently replaced in case of orchestration as the involved web services do not know the actual business process whereas it will be difficult in case of choreography.

Consequently, we notice that orchestration has few more advantages over choreography:

- The coordination of component processes is managed by a centralized known coordinator.
- Web services are used in large business scenarios and they are unaware of that.
- In case faults occur, orchestration can manage alternative scenarios.

So orchestration is preferred for business implementations. In our work, we choose an orchestration process flow to implement our approach.

2.4 Example

In this section, we present an example of composite web services that will be used in other sections to explain our framework. Before introducing this example, we briefly give an overview of state charts, which are used to represent it. A state chart is composed of:

- Filled circle, pointing to the initial state;
- Hollow circle containing a smaller filled circle, indicating the final state (if any);
- Rounded rectangle, denoting a state. This rectangle contains the name of the state;
- Arrow, denoting transition. The name of the event causing this transition labels the arrow body.

These elements and other additional notations are shown in Figure 2.5.

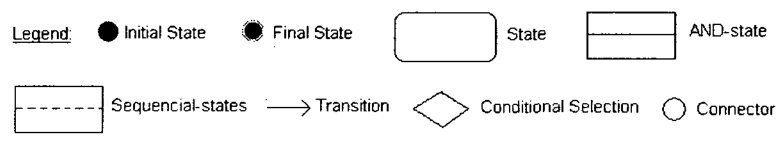


Figure 2.5 State Charts Legend

Figure 2.6 illustrates the state chart of a composite web service process using orchestration: a ticket reservation service. This system is a real-life composite service for travel organization. It is described in BPEL as a state chart. The whole process is composed of states (simple, sequential or and-states). Sequential and and-state states contain other embedded state charts. Initially, the process is in the "*Itinerary Received*" state because the process receives an itinerary from the client. Then the process invokes the airline reservation web service. If the airline reservation system is done without faults, the vehicle and hotel reservations services will be invoked in parallel. If a time-out or fault occurs, the process will end with errors. Otherwise, the invocation of these web services is done correctly. The process moves then to the "*Itinerary Modified*" state. At the end, when the submission is done, the process moves to the "*Itinerary Returned*" state and so, the itinerary is returned to the client.

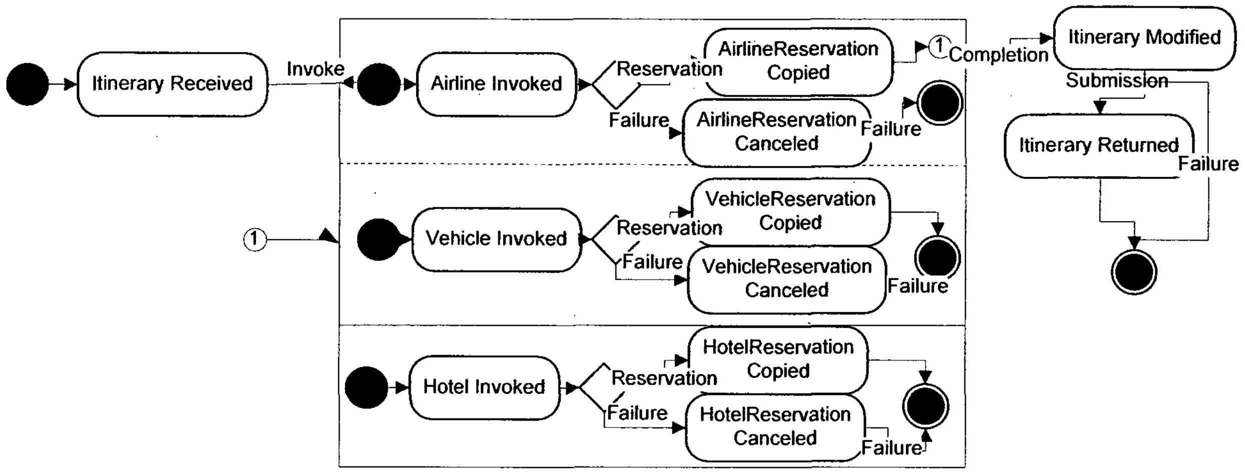


Figure 2.6 State Chart of a Ticket Reservation System

Chapter 3: Model Checking

3.1 Introduction

Given a simplified model of a system and a specific specification, the concept of model checking consists of testing automatically whether this model meets this specification. To go to the root, the original work in the model checking of temporal logic formulas was done by E.M. Clarke and E. A. Emerson [8] [9] [10] and by J. P. Queille and J. Sifakis [33]. Clarke, Emerson, and Sifakis shared the 2007 Turing Award for their work on model checking [32] [36].

Model checking has been used in many real applications, including electrical circuits, digital controllers and communication protocols. Systems are generally hardware or software systems that could have many safety requirements like for example the absence of deadlocks.

The system consists of several components designed to interact with one another and with the system's environment. The system has temporal properties, which will be explained in details in a Section 3.3. The model and the specification should be formulated in a logical language such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL).

The user provides a model of the system and a formulation of the property to be proven. Model checking tool then determines whether or not the model satisfies the property. Therefore, model checking amounts to determining the truth of formulas in models, i.e. whether $M \models \varphi$. However, automatic model checker may have to traverse all reachable system states (state explosion). For this, state space must be finite.

Now, combining composite web services (Chapter 2) and model checking (Chapter 3) is one of the axes of this thesis. Model checking of composite web services has been studied before in [2] [13] [16] [17] using different model checkers. The main difference with our work is in terms of the properties to be verified and the underlying technique. This idea will be developed later on in Chapter 4.

For now, this chapter mainly defines model checking, the system models (Section 3.2), the LTL and CTL properties (Section 3.3), and the verification process (Section 3.4).

3.2 System Models

We will consider the system model to be a Kripke Model [7,8]. A kripke structure *K* is a tuple $K = (S, I, R, Label)$

Where:

- S: a countable set of states
- $I \subseteq S$: a set of initial states
- $R \subseteq S \times S$: a transition relation satisfying $\forall s \in S. (\exists s' \in S. (s, s') \in R)$
- Label: $S \rightarrow 2^{\Phi p}$: an interpretation function where Φp is the set of atomic propositions.

Figure 3.1 shows an example of a Kripke structure [4]:

Where $\Phi p = \{P, Q, R\}$; $S=\{s0, s1, s2, s3, s4\}$, $I=\{s0\}$; $(s0, s1) \in R$, $(s0, s2) \in R$, $(s2, s2) \in R$, $(s3, s4) \in R$; Label $(s0)=\{P, \neg Q, \neg R\}$, Label $(s3)=\{P, \neg Q, R\}$

We have to define paths in these structures. Here are some definitions concerning paths [4]:

- A path Is an infinite sequence of states

$\sigma = s_0\, s_1\, s_2 \ldots$

- Suffix of a path starting at s_i

$\sigma_i = s_i\, s_{i+1}\, s_{i+2} \ldots$

- State in a path: $\sigma[i] = s_i$

- Path(s): the set of paths starting in state s

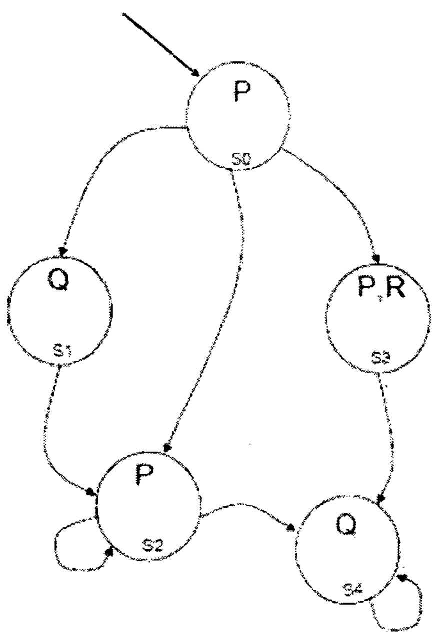


Figure 3.1 Kripke Model Example

Satisfiability and validity are two important concepts in kripke structures.

Satisfiability is when given a formula Φ there exist a Kripke structure K such that K satisfies Φ (i.e. $K \models \Phi$).

Validity is when given a property we have for all Kripke structures K: $K \models \Phi$.

3.3 Properties

The properties to be checked can be written in propositional LTL and CTL format. First, we will introduce the syntax of these two languages. Let Φp be the set of atomic propositions and p ™ Φp.

PLTL syntax is as follows:

$\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid X\Phi \mid \Phi\ U\ \Phi$

CTL syntax is as follows:

$\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid \Phi\ U\ \Phi \mid EX\Phi \mid AX\Phi \mid E(\Phi\ U\ \Phi) \mid A(\Phi\ U\ \Phi)$

$X\Phi$ means in the next state Φ is true. $\Phi\ U\ \Psi$ means Φ is true until Ψ becomes true. *E and A* are the existential and universal quantifiers over paths. *F* (future) and *G* (globally) are abbreviations i.e.:

$F\ \Phi \equiv True\ U\ \Phi$

$G\ \Phi \equiv \neg F \neg\Phi$

Note that LTL is very common in practical model-checking. LTL is used when time is modeled to be linear. However, CTL is used if we want to support branching instead of linear time.

A path is an ordered sequence of states, such that each state is followed by its next state via a transition.

LTL semantics is given as usual using a Kripke structure equipped with a valuation function *L* defined as follows: $L\colon S \times \Phi p \rightarrow \{True,\ False\}$. LTL semantics is as follows (we also give the semantics of some abbreviations for more convenience):

$\sigma \models p$ iff $L(x(0), p) = True$, where $p \in \Phi p$

$\sigma \models \neg\Phi$ iff $x \models \Phi$

$\sigma \models \Phi \wedge \Psi$ iff $x \models \Phi$ and $x \models \Psi$

$\sigma \models \Phi \vee \Psi$ iff $x \models \Phi$ or $x \models \Psi$

$\sigma \models X\Phi$ iff $\sigma(1) \models \Phi$

$\sigma \models G\Phi$ iff for all $i \geq 0$, $\sigma(i) \models \Phi$

$\sigma \models F\Phi$ iff there exists an $i \geq 0$ such that $\sigma(i) \models \Phi$

$\sigma \models \Phi\ U\ \Psi$ iff there exists an $i \geq 0$ such that $\sigma(i) \models \Psi$ and for all $0 \leq j < i$, $\sigma(j) \models \Phi$

Given a state *s* in the Kripke structure, CTL semantics is as follows:

$s \models p$ iff $L(s, p) = True$, where $p \in \Phi p$

$s \models \neg\Phi$ iff not $s \models \Phi$

$s \models \Phi \wedge \Psi$ iff $s \models \Phi$ and $s \models \Psi$

$s \models \Phi \vee \Psi$ iff $s \models \Phi$ or $s \models \Psi$

$s \models EX\Phi$ iff there exists a path $s(0), s(1), \ldots$ such that $s(1) \models \Phi$

$s \models AX\Phi$ iff for all paths $s(0), s(1), \ldots, s(1) \models \Phi$

$s \models EG\Phi$ iff there exists a path $s(0), s(1), \ldots$ such that for all $i \geq 0$, $s(i) \models \Phi$

$s \models AG\Phi$ iff for all paths $s(0), s(1), \ldots$, for all $i \geq 0$, $s(i) \models \Phi$

$s \models EF\Phi$ iff there exists a path $s(0), s(1), \ldots$ such that there exists an $i \geq 0$ such that $s(i) \models \Phi$

$s \models AF\Phi$ iff for all paths $s(0), s(1), \ldots$, there exists an $i \geq 0$, such that, $s(i) \models \Phi$

$s \models E(\Phi\ U\ \Psi)$ iff there exists a path $s(0), s(1), \ldots$ such that, there exists an $i \geq 0$ such that $s(i) \models \Psi$ and for all $0 \leq j < i$, $s(j) \models \Phi$

$s \models A(\Phi\ U\ \Psi)$ iff for all paths $s(0), s(1), \ldots$, there exists an $i \geq 0$ such that $s(i) \models \Psi$ and for all $0 \leq j < i$, $s(j) \models \Phi$

It is important to know that CTL and LTL can express all common safety and liveness properties.

- Safety properties: Nothing "bad" ever happen. They are formalized using state invariants. So the execution never reaches a "bad" state.
- Liveness properties: Something "good" keeps happening. They are formalized using temporal logic. We have special logic for describing sequences.

3.4 Verification Method

The model checking technique consists of computing whether or not a formal model M representing the system satisfies a logical formula φ describing a property. Formally, this problem is denoted by: $M \models \varphi$ or $M \not\models \varphi$. The computation is usually automatic for finite models. The approach used in this work is called symbolic model checking. This approach avoids building or exploring the state space corresponding to the models explicitly. Instead, a symbolic representation is used based on ordered binary decision diagrams (OBDDS) or propositional satisfiability (SAT) solvers [20]. Model checking consists of three parts: A framework for modeling software (some specification language), a specification language for describing properties to be verified and a verification method for establishing if description satisfies the specification.

The model checker we use in this thesis is NuSMV [34]. NuSMV is a software tool for the formal verification of finite state systems based on symbolic model checking. It has been developed jointly by ITC-IRST and Carnegie Mellon University. NuSMV allows checking finite state systems against specifications in the temporal logics LTL and CTL. The input language of NuSMV allows the description of finite state systems that range from completely synchronous to completely asynchronous. The basic purpose of the NuSMV language is to describe the transition relation of a finite Kripke structure. It supports modular hierarchical descriptions and definition of reusable components. This tool has been designed as an open architecture for model checking. It is aimed at reliable verification of industrially sized designs, for use as a backend for other verification tools and as a research tool for formal verification techniques.

The advantage of NuSMV is the flexibility in the use, but sometimes with non expert users there is a danger of inconsistency. To manage this inconsistency, we provide an automatic translation from the Kripe-like structure obtained by the translation procedure from the operational behavior, to the SMV code. The properties to be checked are also extracted from the control behavior and translated into LTL and/or CTL.

To be able to perform verification, we need a modeling language that describes the system, a specification language that formulates the properties and some calculus and algorithm to be able to verify the specification. Then, the model checker checks the properties in the system model and gives the result. The result could be: Yes, the property is satisfied or No with a counterexample. Figure 3.2 shows this model checking approach.

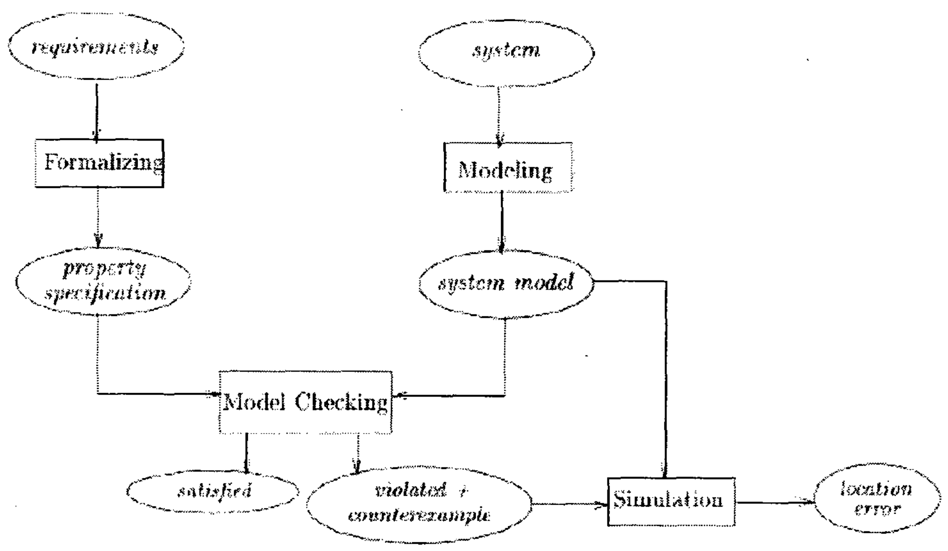


Figure 3.2 Model Checking Approach

In the NuSMV model checker the SMV language is used to describe the system model. Here is a part of SMV syntax [19].

*** Expressions**

Expr ::

```
    atom                ;; symbolic constant
    | number            ;; numeric constant
    | id                ;; variable identifier
    | "!" expr          ;; logical not
    | expr1 <op> expr2
    | "next" "(" id ")"  ;; next value
    | case_expr
    | set_expr
```

*** The Case Expression**

Case_expr :: "case"

expr_a1 ":" expr_b2 ";"

...

expr_an ":" expr_bn ";"

"esac"

- Guards are evaluated sequentially.
- The first one that is true determines the resulting value
- If none of the guards are true, result is numeric value 1

*** State Variables**

Decl :: "VAR"

atom1 ":" type1 ";"

atom2 ":" type2 ";"

...

- State is an assignment of values to a set of state variables
- Type of a variable – boolean, scalar, user defined module, or array.

*** ASSIGN declaration**

Decl :: "ASSIGN"

dest1 ":=" expr1 ";"

dest2 ":=" expr2 ";"

...

Dest :: atom

| "init" "(" atom ")"

| "next" "(" atom ")"

*** Variable Assignments**

- Assignment to initial state:

  init(value) := 0;

- Assignment to next state (transition relation)

  next(value) := value + carry_in mod 2;

- Assignment to current state (invariant)

  carry_out := value & carry_in;

- Either init-next or invar should be used, but not both

- SMV is a parallel assignment language

*** Circular definitions**

- Circular definitions are not allowed

- This is illegal:

  - a := next(b);

    next(b) := c;

    c := a;

- This is accepted:

  - init(a) := 0;

    next(a) := !b;

    init(b) := 1;

    next(b) := !a;

*** Non-determinism**

- Completely unassigned variable can model unconstrained input.

- {val_1, ..., val_n} is an expression taking on any of the given values nondeterministically.
- Nondeterministic choice can be used to model an implementation that has not been refined yet and can be used in abstract behavior

*** ASSIGN and DEFINE**

- VAR a: boolean;

  ASSIGN a := b | c;

  - declares a new state variable a
  - becomes part of invariant relation
- DEFINE d:= b | c;
  - is effectively a macro definition, each occurrence of d is replaced by b | c
  - no extra BDD variable is generated for d
  - the BDD for b | c becomes part of each expression using d

*** SPEC declaration**

- Decl :: "SPEC" ctlform
- Ctlform :: expr ;; bool expression

  | "!" ctlform or (ltlform)

  | ctlform1 <op> ctlform2

  | "E" pathform

  | "A" pathform
- Pathform :: "X" ctlform (or ltlform)

| "F" ctlform (or ltlform)

| "G" ctlform (or ltlform)

| ctlform1(or ltlform1) "U" ctlform2(or ltlform2)

*** Modules and Hierarchy**

- Modules can be instantiated many times, each instantiation creates a copy of the local variables
- Each program has a module main
- Scoping
    - Variables declared outside a module can be passed as parameters
- Parameters are passed by reference.

Chapter 4: Proposed Model

4.1 Introduction

After defining composite web services in Chapter 2 and model checking technique in Chapter 3, we present in this chapter our proposed model.

In this chapter, we focus on the verification issue and we propose a new verification approach, based on formal model checking, for conversations between composite web services. In terms of composition [3] [26], two approaches have been proposed: choreography and orchestration [29] [30] [31]. As explained before, on the one hand, choreography specification identifies the set of allowable conversations for a composite web service. An orchestration, on the other hand, is an executable specification that identifies the steps of execution for the peers. In this work, the focus is on the orchestration only. Furthermore, we consider conversations between web services through their two behaviors: *operational* and *control* [27] [39]. A control behavior describes the general behavior of any process related to composite web services. However, an operational behavior is a behavior specific to each case study according to its business logic. In [27] and [39], these two behaviors have been investigated only for isolated or individual web services out of any composition. In this work, we design the control and operational behaviors for composite web services. Then, we map the control behavior to the operational behavior in order to verify the synchronization in the composition process using model checking technique and assuming that the interaction is controlled by a central coordinating process. Model checking is a formal verification method used to check the correctness of a design model M in terms of the satisfaction of some properties φ, such as safety and liveness. Formally, the problem is to check if $M \models \varphi$ where M is a formal model and φ is a formula expressed in some logics. Model checking has been detailed in Chapter 3.

This chapter is organized as follows. Section 4.2 describes our proposed model. Therefore, in Section 4.2.1 we introduce the formalization and modeling of composite web services. We consider an orchestration model as mentioned before and define the different components of this orchestration. In Section 4.2.2, we present some rules that guarantee a good conversation between web services in a composite setting. In Section 4.2.3, we verify the good synchronization of the conversations among the web services. To do that, we present the control behavior that would be applicable for all the orchestrations of composite web services. Then, we study the operational behavior of a ticket reservation system case study and an ATM case study. At the end, in Section 4.3 we verify the synchronization of the two different types of behaviors, which are the control and operational behaviors using a model checking approach on these two case studies.

4.2 The Proposed Model

4.2.1 Modeling and Formalizing Composite Web Services

Each web service can provide many functionalities, but when it is unable to provide alone a user request, it communicates with other web services either to provide a part of the requested service or to request another part of it. This is the objective of compositions process. The orchestration-based composition of these web services is formally defined as follows.

Definition 1: *The orchestration-based composition of a set of web services is a 4-tuple:* $CW = \langle W, w_0, L, T \rangle$, where:

- W *is the set of web services that interact in the composition;*
- w_0 *is the client web service (the web service that initiates the orchestration process);*
- L *is the set of labels used for the transitions;*
- $T \subseteq W \times L \times W$ *is the set of labeled transitions between the web services.*

Each web service w ™ W is defined as follows.

Definition 2: *A web service is a 5-tuple:* $W = < S, s_0, F, Li, Ti >$

where:

- *S is the set of states that form the behavior of the web service;*
- *s_0 is the initial state of this particular web service;*
- *F is the set of final states;*
- *Li is the set of labels used for the internal transitions;*
- *$Ti \subseteq S \times Li \times S$ is the set of internal labeled transitions inside the web service.*

In fact, a composite web service consists of a set of individual services (or peers), which interact with each other via messages. A conversation is a sequence of messages exchanged among peers participating in a composite web service [6]. Formally, a conversation between n web services is represented as a finite path as follows:

$$w_0 \xrightarrow{a_0} w_1 \ldots \xrightarrow{a_{n-2}} w_{n-1}$$

where $\forall\ 0 \leq i < n-1$ (w_i, a_i, w_{i+1}) ™ T.

To verify if the conversations generated by the composite web service satisfy certain properties, we propose in this work to use model checking, where the desired properties are expressed in a logical language. Precisely, we use symbolic model checking [12] and properties are expressed in two languages: LTL (Linear Temporal Logic) and CTL (Computation Tree Logic). Before introducing the verification method, we define the conversations among web services and their synchronization in the next section.

4.2.2 Synchronization of Conversations among Web Services

To guarantee the correctness of the behavior of the orchestration-based composite web services, their synchronization is needed. To capture this synchronization, we divide the composition behavior into control and operational.

A- Control Behavior

The control behavior for composite scenarios shows the execution progress of a typical orchestration-based composite web service. Such a behavior is supposed to be domain-application independent, so general for all composite services. Its objective is to control the business logic execution as it provides the guidelines for an appropriate composition behavior. Based on the idea of separation of concerns, this general behavior facilitates the reusability of composition scenarios as it is independent from any specific business case. The idea is to design a general control behavior that would be applicable for all the orchestrations of composite web services.

Figure 4.1 depicts the state chart representing the control behavior of the composition scenario. At the initial state, the process is *not activated*, and then when a certain request is sent from a client, the process moves to the *received* state because it receives the request from the client. If any failure occurs, two choices are possible, either the process is *suspended* and we have a retrial, or the process is *aborted* and it ends. When the process reaches the *received* state and no errors occur, it can invoke a certain web service, so the process moves to the *invoked* state, then the web service replies, so the process returns to the *received* state or it can do some other processing so it moves to the *processing* state then to the *received* state. The process can send other requests to other web services and so on. At the end, the process receives the final information and it has to commit the action back to the client. When the commitment is satisfied, the process moves to the *done* state. At any time, if an error occurs, the process can be diverted to the *suspended* or *aborted* states. *Compensated* state could be reached after failed retrials, so the process goes back to the *not-activated* state, or after the commitment.

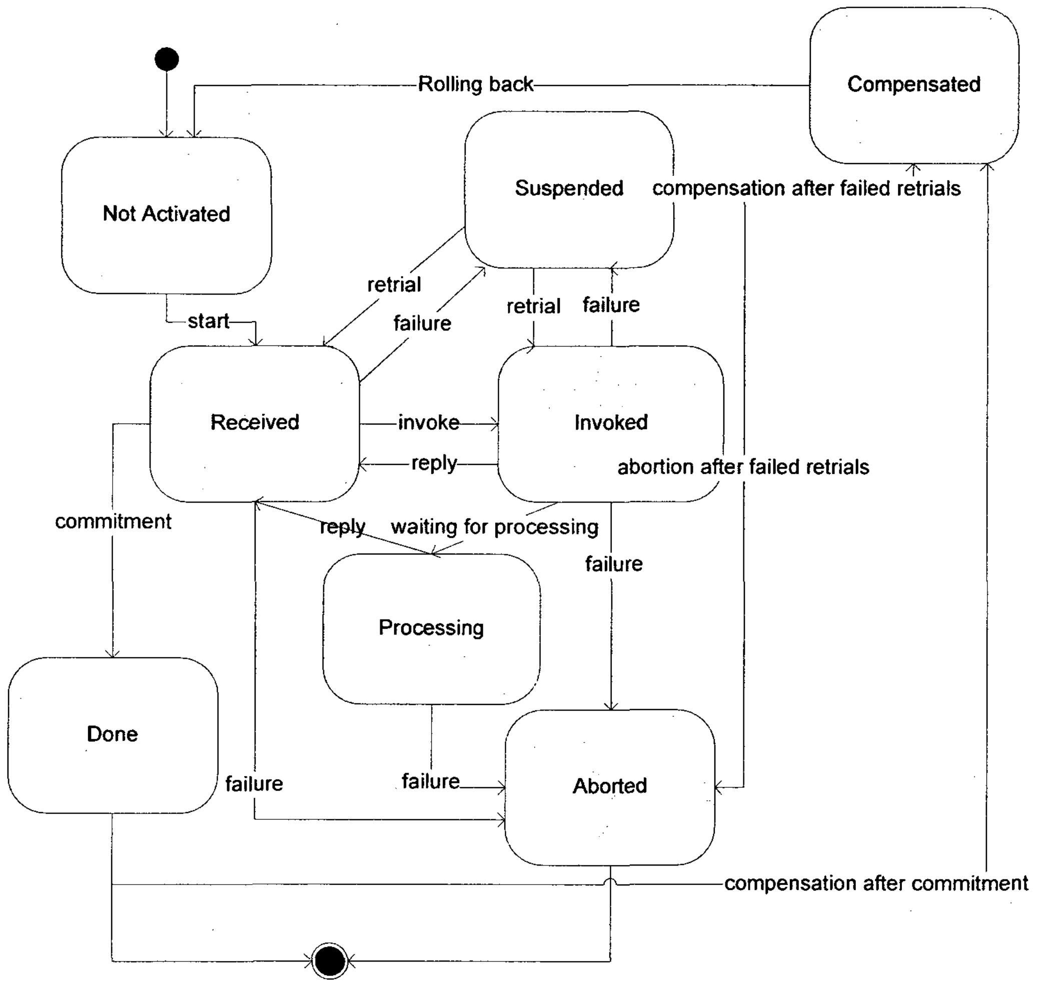


Figure 4.1 Control Behavior of a Composite Web Service

B- Operational Behavior

The operational behavior of a composite web service shows the business logic describing the functioning of a given orchestration-based composition. Unlike the control behavior, the operational behavior is domain-application dependant. This behavior is supposed to be overseen by the control behavior. The conformance to the control behavior (in terms of synchronization) is a proof that the operational behavior is well designed.

To explain this notion, we consider first here a concrete example: a ticket reservation service. The state chart in Figure 4.2 illustrates the operational behavior of this composite web service process using orchestration. This state chart was previously explained in Section 2.4.

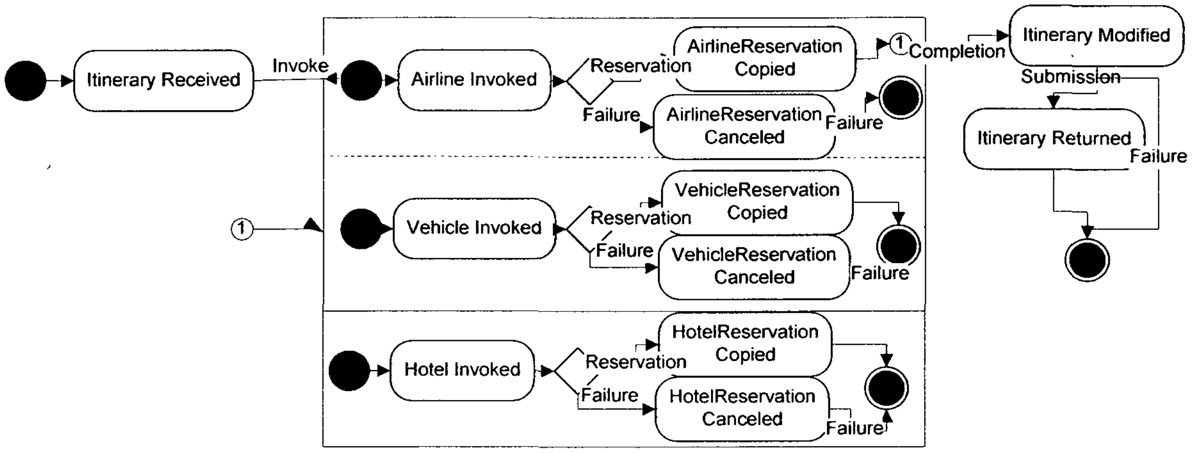


Figure 4.2 Operational Behavior of a Ticket Reservation System

Figure 4.3 illustrates another example of operational behavior of the composite web service process using orchestration: An ATM system. In this figure, we also consider an orchestration case of composite web services. In an ATM system, the user must enter the card first, then enter the PIN, then the system will invoke the bank system to check if the login info are correct. If they are wrong, we will have a failure and the process will end. If they are correct, the user will have choices of transactions to choose from. The user can choose to withdraw, deposit or check balance.

- If the user chooses to withdraw, again the bank system will be invoked to check if the user has enough balance to withdraw from, if not the process ends with failure. If it has enough balance, then the withdrawal will be done and then the user can choose to print a receipt or not. If not he logouts and the process ends. If he wants a receipt, another system will be invoked: the printer system and the receipt will be printed so the user will receive the receipt of his transaction.

- If the user chooses to deposit, the bank system is invoked again and the amount he enters will be checked to be in a certain limit. And the deposit will be done if the amount is ok; if not the process will end with a failure. Now, if the deposit is done correctly, the user also will have a choice to have a receipt. So, if he wants a receipt, another system will be invoked: the printer system and the receipt will be printed so the user will receive the receipt of his transaction.
- If the user chooses to get his balance, the bank system is invoked again and the balance is displayed for the user. Here, also the user will have a choice to have a receipt. So, if he wants a receipt, another system will be invoked: the printer system and the receipt will be printed so the user will receive the receipt of his transaction.

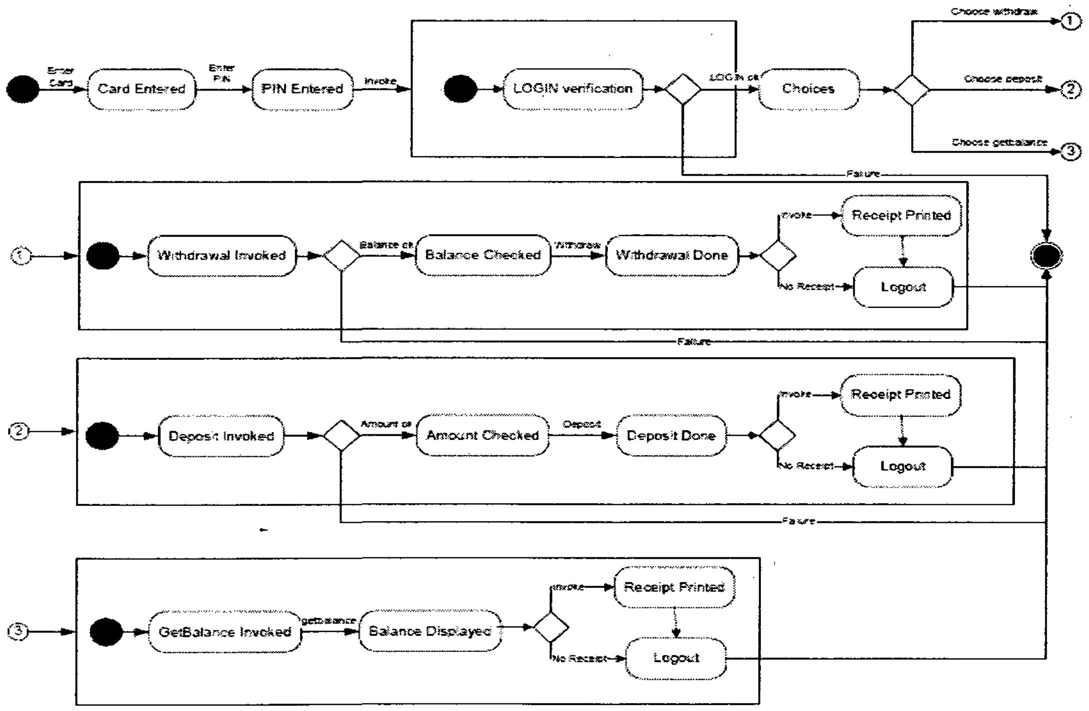


Figure 4.3 Operational Behavior of an ATM System

4.2.3 Verification

To verify the synchronization of the two different types of behaviors, we convert the operational behaviors into a system model represented as a Kripke structure, and we extract from the control behavior all the properties available in a temporal logic format. By doing this, we can verify, using model checking, that the operational behavior is conform to the control behavior, from which we extract the properties. Thus, soundness and completeness of a composition can be defined as follows.

Definition 3: *An orchestration of composite web services is sound and complete iff all the properties of the control behavior are satisfied in the operational behavior system model (soundness) and vice-versa (completeness).*

A- Properties to be checked

We have explained in the previous chapter the LTL and CTL syntaxes in details.
Now, in LTL, the default path quantifier is A, so a state satisfies a formula if it is satisfied in *all paths* starting by this state. The reason behind using two different languages LTL and CTL to specify the properties is because they are not equivalent. There are properties that can be expressed in LTL but cannot be expressed in CTL (for example $AF(p \wedge Xp)$) and vice versa (for example: $AG(EFp)$). We also notice that we are considering *fair* LTL and CTL [12], which means in any computation, some states, called *fair states*, should be reached. In the control behavior depicted in Figure 2.2, *Done*, *Aborted* and *Compensated* are fair states. Thus, in any execution, *Done* or *Aborted* or *Compensated* should be reached.

To specify the properties we aim to check from the control behavior, we will consider the following initials (see Figure 2.2): Not activated: *Na* / Received: *Re* / Invoked: *In* / Suspended: *Su* / Aborted: *Ab* / Processed: *Pr* / Compensated: *Co* / Done: *Do* / End: *En*

Let $\rightarrow$ be the logical implication. Examples of fair LTL properties we can verify as extracted from the control behavior are:

1- $\Phi = G(Na \rightarrow XRe)$

2- $\Phi = G(Re \rightarrow XF(In \vee Ab \vee Su \vee Do))$

3- $\Phi = G(Co \rightarrow XFNa)$

4- $\Phi = G(Do \rightarrow XF(En \vee Co))$

5- $\Phi = G((Do \vee Ab) \rightarrow XFEn)$

6- $\Phi = G(In \rightarrow XF(Ab \vee Pr \vee Re \vee Su))$

Explaining for example the first property, we always have after a *non-activated* state a *receive* state. The second property states that always after a *receive* state, we have an *invoked*, an *aborted*, a *suspended*, or a *done* state in the future. In the fifth property, we always have an *end* state after an *aborted* or *done* state.

Examples of CTL properties from the control behavior are:

1- $\Phi = AG(Na \rightarrow AXRe)$

2- $\Phi = AG(\mathrm{Re} \rightarrow AXAF(In \vee Ab \vee Su \vee Do))$

3- $\Phi = AG(Co \rightarrow AXAFNa)$

4- $\Phi = AG(Do \rightarrow AXAF(En \vee Co))$

5- $\Phi = AG((Do \vee Ab) \rightarrow AXAFEn)$

6- $\Phi = AG(In \rightarrow \mathrm{AXAF}\ (Ab \vee Pr \vee Re \vee Su))$

Examples of properties in CTL that cannot be expressed in LTL are:

7- $\Phi = AGEF(En)$

8- $\Phi = AGEF(Do)$

9- $\Phi = AGEF(Ab \vee Do)$

10- $\Phi = AGEF(Re \rightarrow In)$

11- $\Phi = \mathrm{AGEF}((In \rightarrow EXPr) \vee (In \rightarrow EXRe))$

12- $\Phi = AGEF((\mathrm{Pr} \rightarrow EXRe) \vee (\mathrm{Pr} \rightarrow EXAb))$

Property 7 states that in all paths there always exists a path where in the future we have and end state. Property 10 states that in all paths there always exists a path where a receive state should be followed by an invoke state.

B- System Model

After extracting the properties to be checked from the control behavior, the second step in the verification process is to build the Kripke-like model from the operational behavior. The resulting model is the one we use to automatically generate the SMV code used by the NuSMV model checker [7]. This translation is automatic and is as follows. Each state s_{op} in the operational behavior is translated to a set of states and transitions in the Kripke-like structure M and each transition is translated to one or many transitions. If s_{op} is a simple state, it is translated into one state in M with the same content. If s_{op} is a state chart, then two cases are possible: 1) the state is a sequential state; 2) the state is an and-state. In both cases, each simple state is translated into one state with the same content and all the end states are translated to one end state. In the first case, the connector is replaced by the next state if this state is simple, or by the first state of the next sequential state or and-state. In the second case, the and-states are simply considered as sequential and the sequence order is selected randomly. The reason is that in an and-state, all the states should be considered but the order of this consideration is not important. Only the last state in the selected order is related to the next state by a transition. The number of possible Kripke-like structures depends then on the number of states in and-states. However, all the executions are equivalent, which means that only one structure should be considered. The conditional selections are simply ignored as they are captured by deterministic transitions. Transitions between simple states are translated to transitions between the corresponding states in the Kripke-like structure.

Transitions between simple and sequential states or and-states are translated into transitions between the corresponding state of the simple state and the corresponding state of the first state of the sequential state or and-state.

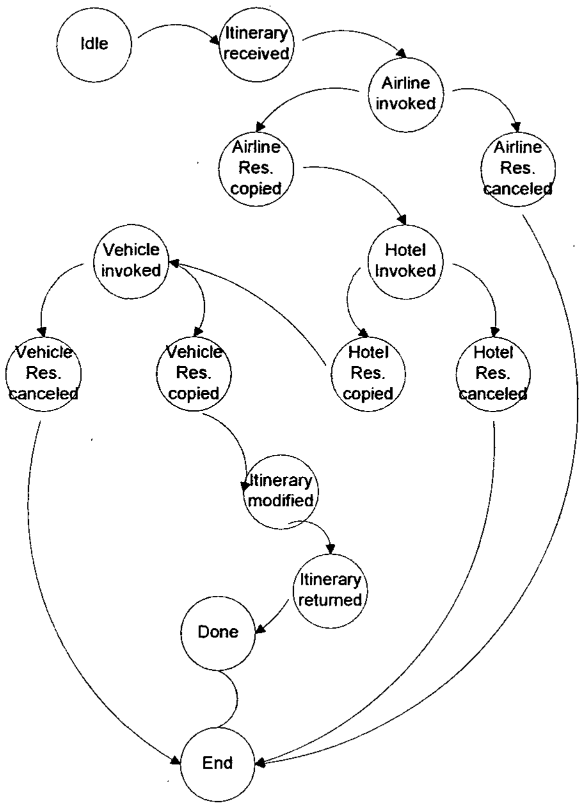


Figure 4.4 Model of the Ticket Reservation Composite Web Service

Figure 4.4 shows the Kripke-like model obtained after translating the operational behavior given in Figure 4.2 (ticket reservation service) using this translation procedure. As illustrated in Figure 4.4, after an airline web service is invoked, the action could be committed directly, or a vehicle and hotel web services could be also invoked depending on the initial client request. At any time the reservation could be canceled and the process is aborted in that case. The atomic propositions that are true in the obtained states using the evaluation function L are those used in the control behavior. Figure 4.5 shows the final Kripke-like model where: $R = Re$, $I = In$, $S = Su$,

$A = Ab$, $P = Pr$, $C = Co$, $D = Do$, and $E = En$. Note that the *idle* state corresponds to a *non-activated* state.

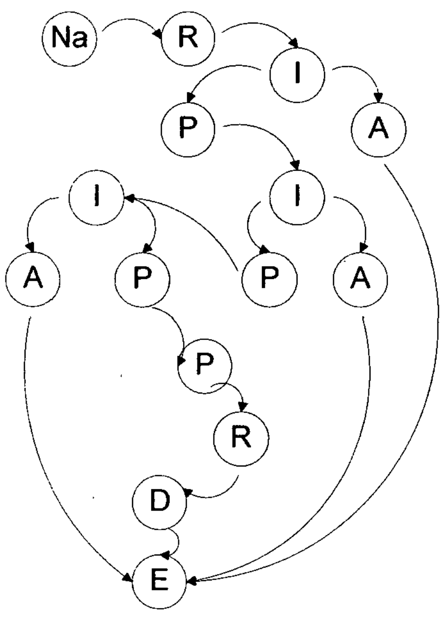


Figure 4.5 Kripke-like Model of the Ticket Reservation Composite Web Service

C- Model Checking Technique

The model checking technique consists of computing whether or not a formal model M representing the system satisfies a logical formula φ describing a property. Formally, this problem is denoted by: $M \models \varphi$ or $M \not\models \varphi$. The computation is usually automatic for finite models. The approach used in this work is called symbolic model checking. This approach avoids building or exploring the state space corresponding to the models explicitly. Instead, a symbolic representation is used based on ordered binary decision diagrams (OBDDS) or propositional satisfiability (SAT) solvers [12].

The model checker we use is NuSMV [7]. We explained about NuSMV in Chapter 3. The advantage of NuSMV is the flexibility in the use, but sometimes with non expert users there is a danger of inconsistency. To manage this inconsistency, we provide an automatic translation from the Kripe-like structure obtained by the translation procedure from the operational behavior, to the SMV code. The properties to be checked are also extracted from the control behavior and translated into LTL and/or CTL. The approach of the model checking is described in Figure 4.6.

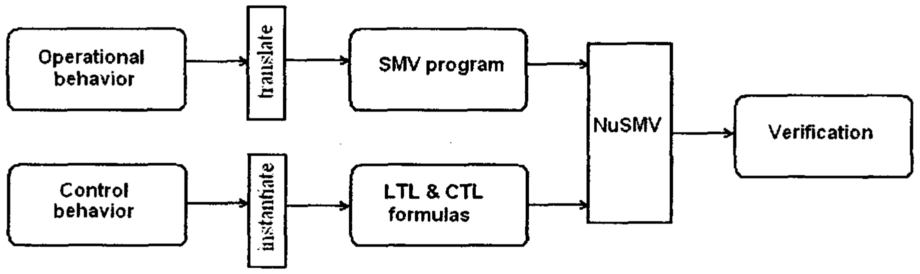


Figure 4.6 Model Checking of Composite Web Services

4.3 Case Studies

Let us continue the example provided in Figures 10 (control behavior) and 11 (operational behavior). First, we use a reduction algorithm like the one used to reduce OBDDS [7] in order to reduce the Kripke-like model illustrated in Figure 4.5. The idea is to reduce the number of states and transitions based on the fact that two states labeled with the same atomic propositions using the valuation function L are equivalent, so they can be reduced to only one state. The transitions are then reduced as follows:

For all s_1 and s_2, if s_2 is reduced to s_1, then:

a. If (s_1, s_2) and (s_2, s_1) are two transitions, then they are replaced by one transition (s_1, s_1);

b. If only one of the two transitions does exist, then it is removed;

c. For all x, if (s_x, s_2) is a transition, then it is removed and replaced by the transition (s_x, s_1) if such a transition does not exist;

d. For all y, if (s_2, s_y) is a transition, then it is removed and replaced by the transition (s_1, s_y) if such a transition does not exist;

Proposition 1: Let K be a Kripke-like model and K' be the reduced model obtained using the reduction algorithm. K and K' are semantically equivalent.

Proof

Let T_K be the set of transitions in K and $T_{K'}$ be the set of transitions in K'. To prove the proposition, we should prove that for each transition in T_K there is a semantically corresponding transition in $T_{K'}$ (soundness) and vise-versa (completeness).

We prove soundness by deduction on the reduction rules. For the first rule, the removed transitions from T_K are semantically captured by the loop transition in $T_{K'}$ as the two states s_1 and s_2 are equivalent. For the second rule, the removed transition is captured by the state. In fact, here we have (s_1, s_2) ™ T_K and s_1 and s_2 are equivalent, so one state and the transition are redundant. For the third and fourth rules, the removed transitions are captured by the replaced transitions because (s_x, s_1) and (s_x, s_2) are equivalent and (s_1, s_y) and (s_2, s_y) are equivalent since s_1 and s_2 are equivalent.

The completeness is simply proved by construction as all the transitions in K' are constructed from the transitions in K.

The reduction algorithm preserves then the semantics and is automatically performed. Figure 4.7 depicts the result of reducing the Kripke-like model presented in Figure 4.5.

Then, the reduced model is automatically translated to the SMV code used by NuSMV model checker. SMV code mainly describes the transition relation of the Kripke-like model (Figure 4.8).

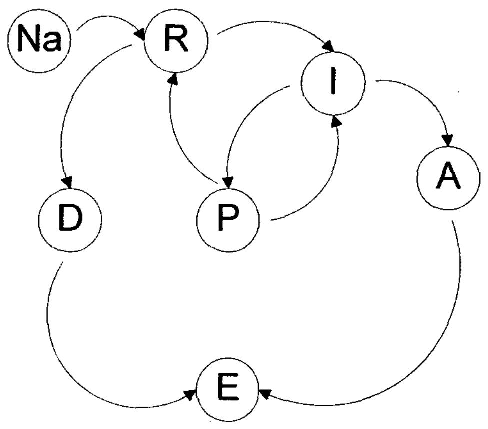


Figure 4.7 Reduced Kripke-like Model of the Ticket Reservation Composite Web Service

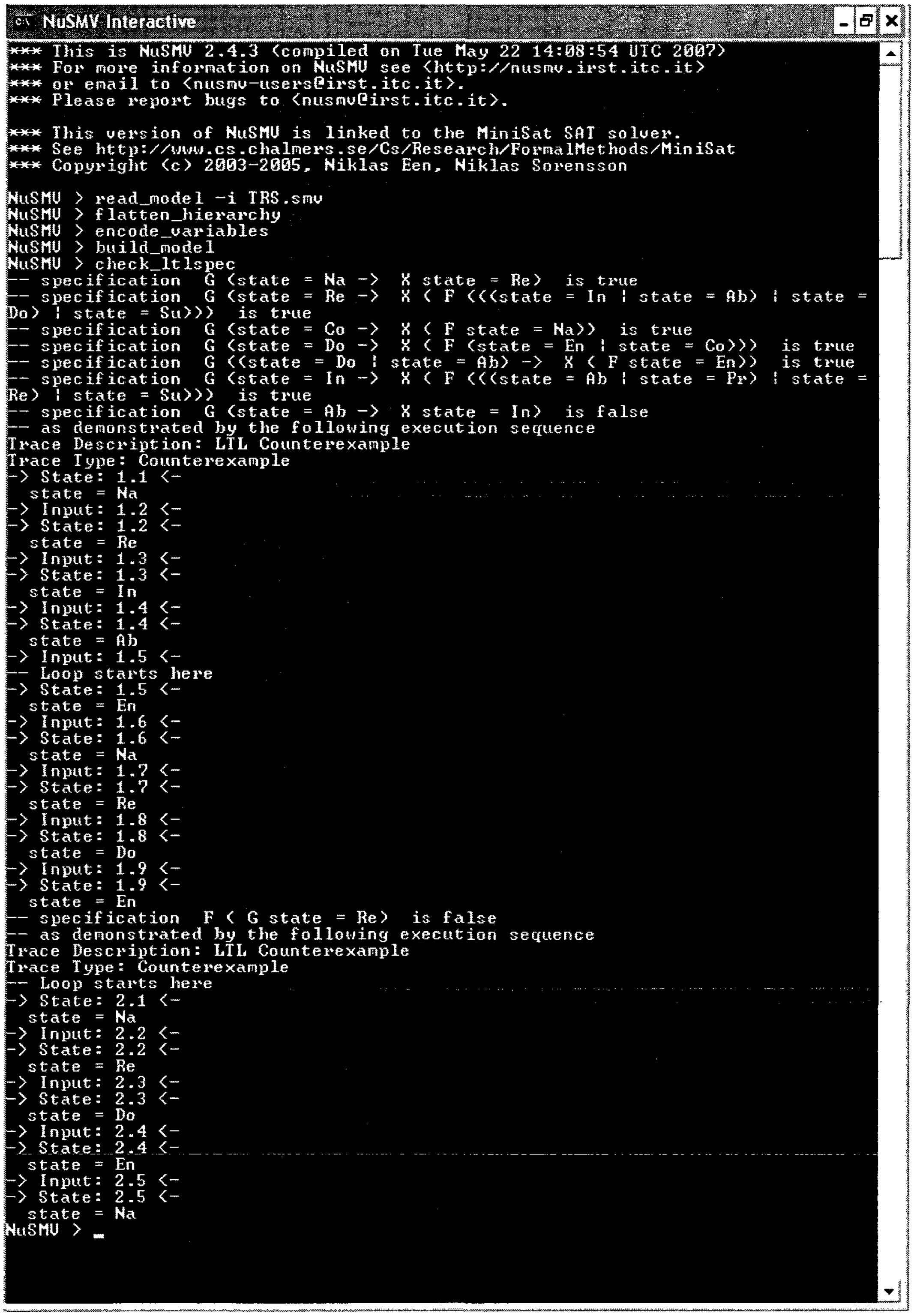
To check the properties described in Section 4.2.3, the following commands are used:

```
NuSMV > read_model –i TRS.smv   (TRS.smv is the name of the smv file we created)
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
NuSMV > check_ltlspec (to check ltl specifications)
NuSMV > check_ctlspec (to check ctl specifications)
```

Figure 4.9 and Figure 4.10 show the result of the model checking procedure (LTL and CTL specifications). First we have to read the .smv program then flatten the hierarchy, encode the variables and build the model. Then, the specifications are checked. For the LTL specifications checking, all the properties are satisfied, except for the last two, for which counter examples are provided (Figure 4.9). For CTL, all the properties are satisfied (Figure 4.10).

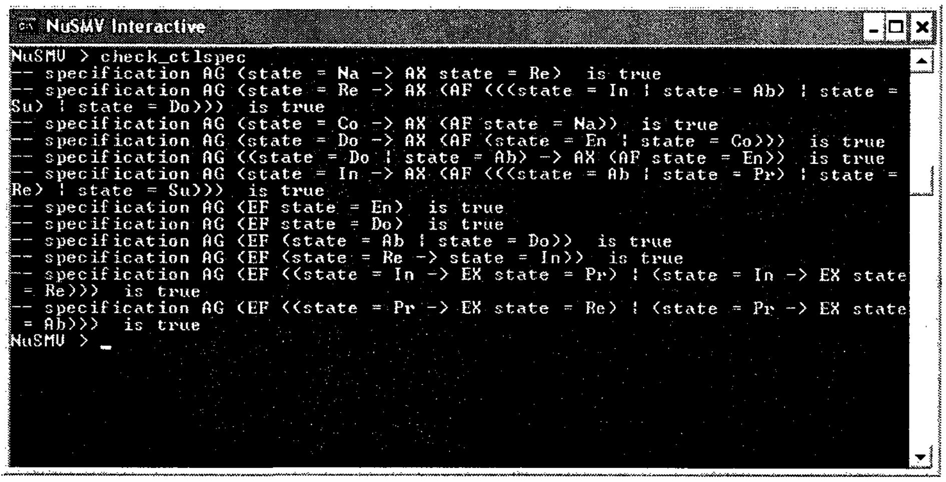
```
MODULE main
VAR
state:{Na,Re,In,Ab,Pr,Do,En,Co,Su};
ASSIGN
init(state):=Na;
next(state):=
        case
           (state=Na):{Re};
           (state=Re):{In,Do};
           (state=In):{Pr,Ab};
           (state=Ab):{En};
           (state=Pr):{In,Re,Pr};
           (state=Do):{En};
           (state=En):{Na};
            1:state;
esac;

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))

--wrong LTL specification
LTLSPEC G (state=Ab -> X state=In)
LTLSPEC F G state=Re

-- CTL Specifications
SPEC AG (state=Na -> AX state=Re)
SPEC AG (state=Re -> AX AF (state=In | state=Ab | state=Su | state=Do))
SPEC AG (state=Co -> AX AF state=Na)
SPEC AG (state=Do -> AX AF (state=En | state=Co))
SPEC AG ((state=Do | state=Ab) -> AX AF state=En)
SPEC AG (state=In -> AX AF (state=Ab |state=Pr | state=Re | state=Su))

SPEC AG EF (state=En)
SPEC AG EF (state=Do)
SPEC AG EF (state=Ab|state=Do)
SPEC AG EF (state=Re -> state=In)
SPEC AG EF ((state=In -> EX state=Pr)|(state=In -> EX state=Re))
SPEC AG EF ((state=Pr -> EX state=Re)|(state=Pr -> EX state=Ab))
```

Figure 4.8 SMV Code for NuSMV Model Checker



```
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

NuSMV > read_model -i TRS.smv
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
NuSMV > check_ltlspec
-- specification  G (state = Na ->  X state = Re)  is true
-- specification  G (state = Re ->  X ( F (((state = In | state = Ab) | state =
Do) | state = Su)))  is true
-- specification  G (state = Co ->  X ( F state = Na))  is true
-- specification  G (state = Do ->  X ( F (state = En | state = Co)))  is true
-- specification  G ((state = Do | state = Ab) ->  X ( F state = En))  is true
-- specification  G (state = In ->  X ( F (((state = Ab | state = Pr) | state =
Re) | state = Su)))  is true
-- specification  G (state = Ab ->  X state = In)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = Na
-> Input: 1.2 <-
-> State: 1.2 <-
  state = Re
-> Input: 1.3 <-
-> State: 1.3 <-
  state = In
-> Input: 1.4 <-
-> State: 1.4 <-
  state = Ab
-> Input: 1.5 <-
-- Loop starts here
-> State: 1.5 <-
  state = En
-> Input: 1.6 <-
-> State: 1.6 <-
  state = Na
-> Input: 1.7 <-
-> State: 1.7 <-
  state = Re
-> Input: 1.8 <-
-> State: 1.8 <-
  state = Do
-> Input: 1.9 <-
-> State: 1.9 <-
  state = En
-- specification  F ( G state = Re)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 2.1 <-
  state = Na
-> Input: 2.2 <-
-> State: 2.2 <-
  state = Re
-> Input: 2.3 <-
-> State: 2.3 <-
  state = Do
-> Input: 2.4 <-
-> State: 2.4 <-
  state = En
-> Input: 2.5 <-
-> State: 2.5 <-
  state = Na
NuSMV > _
```

Figure 4.9 Verification Results using NuSMV Model Checker (LTL Specifications)



```
NuSMV > check_ctlspec
-- specification AG (state = Na -> AX state = Re)  is true
-- specification AG (state = Re -> AX (AF (((state = In | state = Ab) | state =
Su) | state = Do)))  is true
-- specification AG (state = Co -> AX (AF state = Na))  is true
-- specification AG (state = Do -> AX (AF (state = En | state = Co)))  is true
-- specification AG ((state = Do | state = Ab) -> AX (AF state = En))  is true
-- specification AG (state = In -> AX (AF (((state = Ab | state = Pr) | state =
Re) | state = Su)))  is true
-- specification AG (EF state = En)  is true
-- specification AG (EF state = Do)  is true
-- specification AG (EF (state = Ab | state = Do))  is true
-- specification AG (EF (state = Re -> state = In))  is true
-- specification AG (EF ((state = In -> EX state = Pr) | (state = In -> EX state
 = Re)))  is true
-- specification AG (EF ((state = Pr -> EX state = Re) | (state = Pr -> EX state
 = Ab)))  is true
NuSMV > _
```

Figure 4.10 Verification Results using NuSMV Model Checker (CTL specifications)

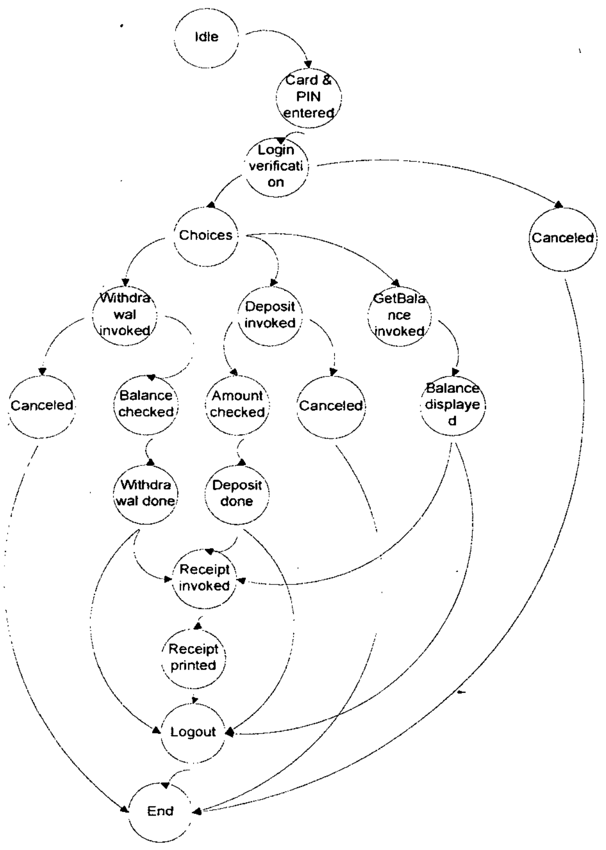


Figure 4.11 Model of the ATM Composite Web Service

Let us consider the second case study that consists of an ATM system. In Figure 4.3 we presented the operational behavior of this system. Now, if we want to follow the same steps of the first case study, we will first build the Kripke-like model from the operational behavior. Figure 4.11 shows the model obtained after its translation from operational behavior. Figure 4.12 shows the final Kripke-like model.

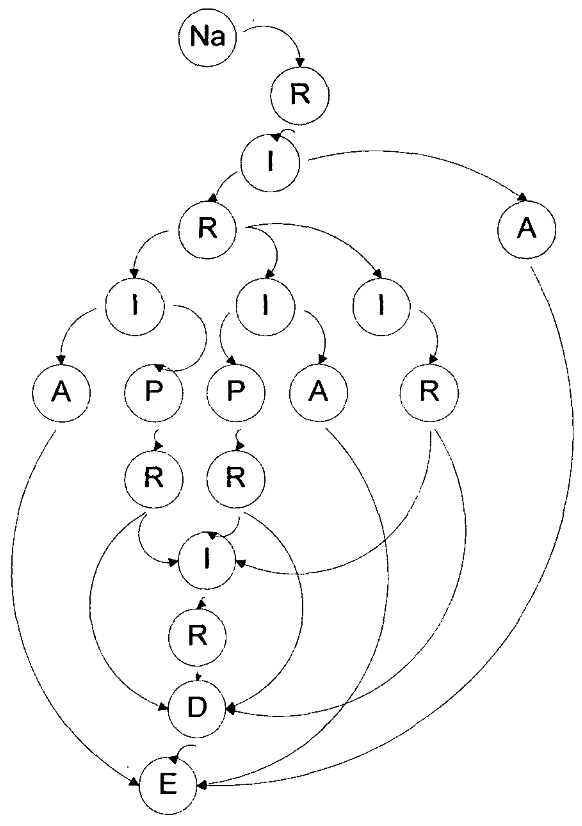


Figure 4.12 Kripke-like Model of the ATM Composite Web Service

We then use the reduction algorithm described earlier in this section that preserves the semantics. Figure 4.13 depicts the result of reducing the Kripke-like model of the ATM presented in Figure 4.12

Then, the reduced model is automatically translated to the SMV code used by NuSMV model checker. SMV code mainly describes the transition relation of the Kripke-like model (Figure 4.14).

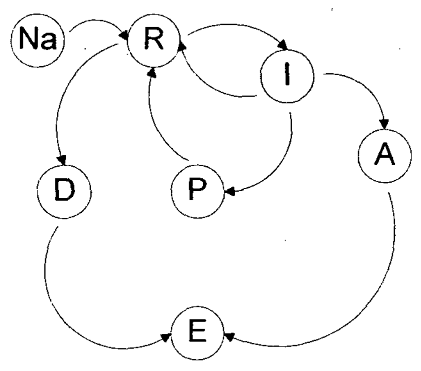
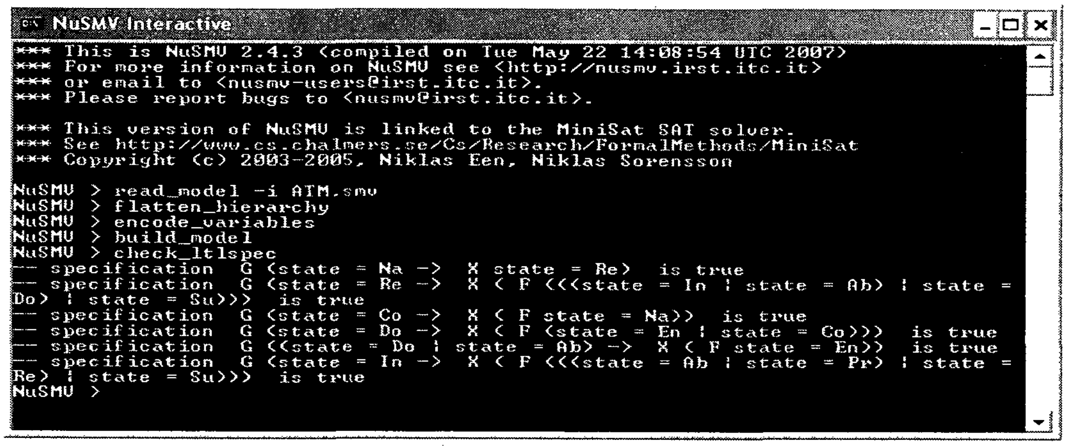


Figure 4.13 Reduced Kripke-like Model of the ATM Composite Web Service

```
MODULE main
VAR
state:{Na,Re,In,Ab,Pr,Do,En,Co,Su};
ASSIGN
init(state):=Na;
next(state):=
    case
       (state=Na):{Re};
       (state=Re):{In,Do};
       (state=In):{Re,Ab,Pr};
       (state=Ab):{En};
       (state=Pr):{Re};
       (state=Do):{En};
       (state=En):{Na};
        1:state;
esac;

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))


-- CTL Specifications
SPEC AG (state=Na -> AX state=Re)
SPEC AG (state=Re -> AX AF (state=In | state=Ab | state=Su | state=Do))|
SPEC AG (state=Co -> AX AF state=Na)
SPEC AG (state=Do -> AX AF (state=En | state=Co))
SPEC AG ((state=Do | state=Ab) -> AX AF state=En)
SPEC AG (state=In -> AX AF (state=Ab | state=Pr | state=Re | state=Su))

SPEC AG EF (state=En)
SPEC AG EF (state=Do)
SPEC AG EF (state=Ab | state=Do)
SPEC AG EF (state=Re -> state=In)
SPEC AG EF ((state=In -> EX state=Pr)|(state=In -> EX state=Re))
SPEC AG EF ((state=Pr -> EX state=Re)|(state=Pr -> EX state=Ab))
```

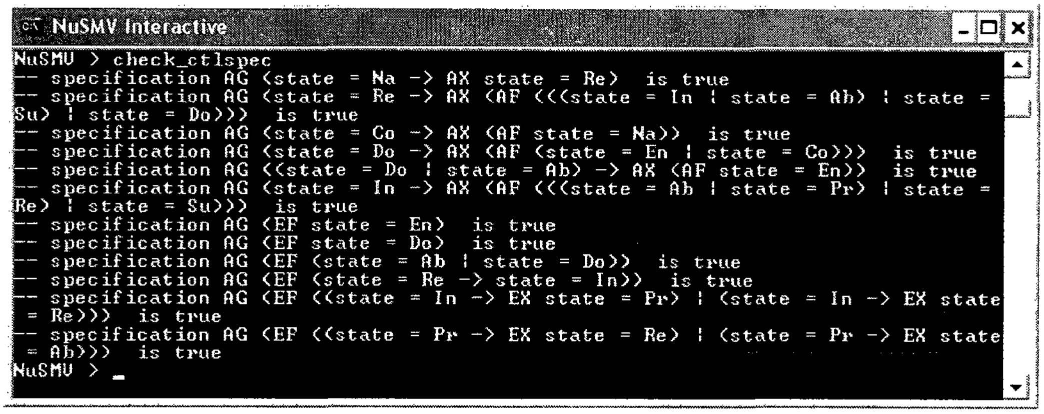
Figure 4.14 SMV Code of ATM System for NuSMV Model Checker

Figure 4.15 and Figure 4.16 show the result of the model checking procedure (LTL and CTL specifications).



```
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

NuSMV > read_model -i ATM.smv
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
NuSMV > check_ltlspec
-- specification  G (state = Na ->  X state = Re)  is true
-- specification  G (state = Re ->  X ( F (((state = In | state = Ab) | state =
Do) | state = Su)))  is true
-- specification  G (state = Co ->  X ( F state = Na))  is true
-- specification  G (state = Do ->  X ( F (state = En | state = Co)))  is true
-- specification  G ((state = Do | state = Ab) ->  X ( F state = En))  is true
-- specification  G (state = In ->  X ( F (((state = Ab | state = Pr) | state =
Re) | state = Su)))  is true
NuSMV >
```

Figure 4.15 Verification Results using NuSMV Model Checker on ATM System (LTL Specifications)



```
NuSMV > check_ctlspec
-- specification AG (state = Na -> AX state = Re)  is true
-- specification AG (state = Re -> AX (AF (((state = In | state = Ab) | state =
Su) | state = Do)))  is true
-- specification AG (state = Co -> AX (AF state = Na))  is true
-- specification AG (state = Do -> AX (AF (state = En | state = Co)))  is true
-- specification AG ((state = Do | state = Ab) -> AX (AF state = En))  is true
-- specification AG (state = In -> AX (AF (((state = Ab | state = Pr) | state =
Re) | state = Su)))  is true
-- specification AG (EF state = En)  is true
-- specification AG (EF state = Do)  is true
-- specification AG (EF (state = Ab | state = Do))  is true
-- specification AG (EF (state = Re -> state = In))  is true
-- specification AG (EF ((state = In -> EX state = Pr) | (state = In -> EX state
 = Re)))  is true
-- specification AG (EF ((state = Pr -> EX state = Re) | (state = Pr -> EX state
 = Ab)))  is true
NuSMV > _
```

Figure 4.16 Verification Results using NuSMV Model Checker on ATM System (CTL specifications)

4.4 Related Work

The concept of control and operational behaviors was previously studied in [27] and [39]. In these two publications, the control behavior illustrates the business logic that underpins the functioning of an isolated web service, and the operational behavior regulates the execution progress of this control behavior by stating the actions to carry out and the constraints to put on this progress. However, the composition and verification aspects were not investigated. The composition issue from a formal perspective and the tools used were also stated in some papers. In [22], Hull et al. describe concepts and assumptions on current work on service composition. They present several composition models including semantic web services, the "Roman" model, and the Mealy conversation model. They also give techniques for analyzing web services such as translating them into formalisms that are suitable for analysis, for example state machines, extended mealy machines, and process algebra. However, synchronization between behaviors and verification of composition design were not analyzed.

Other projects that use model checking techniques for BPEL composite web services verification were done. In [14], Foster et al. verify mediated composite services specified in BPEL against the design specified using Message Sequence Chart and Finite State Process notations. Unlike our proposal, the focus is on the control flow logic and not on the conversations between the composite services. Also, the proposed verification method is not implemented. In [15], the tool presented can be used to check that composite web services satisfy LTL properties. The input of the tool is BPEL specifications that are translated into guarded automata. These automata are then translated to Promela language to check them in the SPIN model checker. This allows the authors to verify designs at a more detailed level and to check properties about message content. Although the verification approach is similar to ours, there are many differences between the two works. In our proposal, the verification is based on separating behaviors and not only on BPEL. Also, the model checking technique we use is different as SMV and NuSMV are based on symbolic model checking and not on automata model checking like in Promela and Spin. Symbolic model checking has an advantage over automata-based technique as it does not suffer from the state explosion problem. Finally, in our proposal, we can check not only LTL specifications like in [15], but also CTL specifications.

In [35], the authors show the importance of asynchronous messaging in sharing information and resources in the form of web processes. Web service interaction models are formalized into a conversation concept with ordering constraints on messages. FIFO queues are considered in the design of message passing between services. In terms of verification, only some abstract strategies of model checking service composition for both bottom-up and top-down design approaches are outlined. However, no analysis or implementation of these strategies is provided. Model checking of composite web services has been studied also in [2] [13] [16] [17] using different model checkers. The main difference with our work is in terms of the properties to be verified and the underlying technique. To the best of our knowledge, this work is the first investigation on separating concerns in composite scenarios and automatically verifying the operational behavior against the control specification using both LTL and CTL languages. The technique is based on analyzing the two behaviors and extracting properties from the general control behavior to be verified in the model represented by the operational behavior of the system. This method enables us to control the orchestration process of the composition in web services and to verify the synchronization of messages between different web services.

In terms of web services interactions, some researchers have studied feature interactions in order to model and monitor undesirable interactions [35] [37]. Feature interactions for web services are described as the situations where the requirements of services are inconsistent [1]. Feature interactions are often seen as the result of complex behavior interleaving for the state machines that represent the features. In [24], a first-order logic model-checking tool called Alloy is used for automated detection of feature interactions. Our proposal is different from this work since we are considering not only undesirable interactions, but all possible interactions that can be extracted from the control behavior. The model checking technique we are using is also different from the first order model checking.

Chapter 5: Implementation

5.1 Introduction

In the previous chapter, we presented our verification approach of messages synchronization among web services. First, we discussed how composite web services could be designed and modeled based on their control and operational behaviors. The operational behavior shows the business logic of the process functionality for a composite web service. The control behavior shows the constraints and states that the operational behavior should be in. Synchronizing both behaviors is a key issue in designing good conversations between the different web services that participate in composite web services. We used symbolic model checking as the verification approach. The properties to be checked are taken from the control behavior and are verified in the different operational scenarios.

In Chapter 4, we translated manually the state charts to SMV code following these steps:

a- Translating the state chart to a model

b- Finalizing the model to a Kripke-like model

c- Reducing this Kripke-like model

d- Translating the reduced model into SMV syntax.

In this chapter, we want to make the translation easy and automatic for the user. We created for that reason an interface where the user enters the different transitions of states existing in the original state chart. The user will also add the LTL and CTL properties and then the SMV file is directly created.

In Section 5.2 we will explain the framework of this SMV Converter in more details. In Section 5.3 we will show a step by step example of the ticket reservation system composite web service. Some samples of the code are listed in Appendix 1.

5.2 SMV Converter

As stated in the previous section, the SMV converter is responsible of converting state charts to SMV code to automatically help the user in the verification process. The SMV converter we created (see Figure 5.1) is composed of four areas. The first one is for state charts. The second and third areas are for LTL and CTL specification. The last area displays the SMV code that will be put in the ".smv" file.

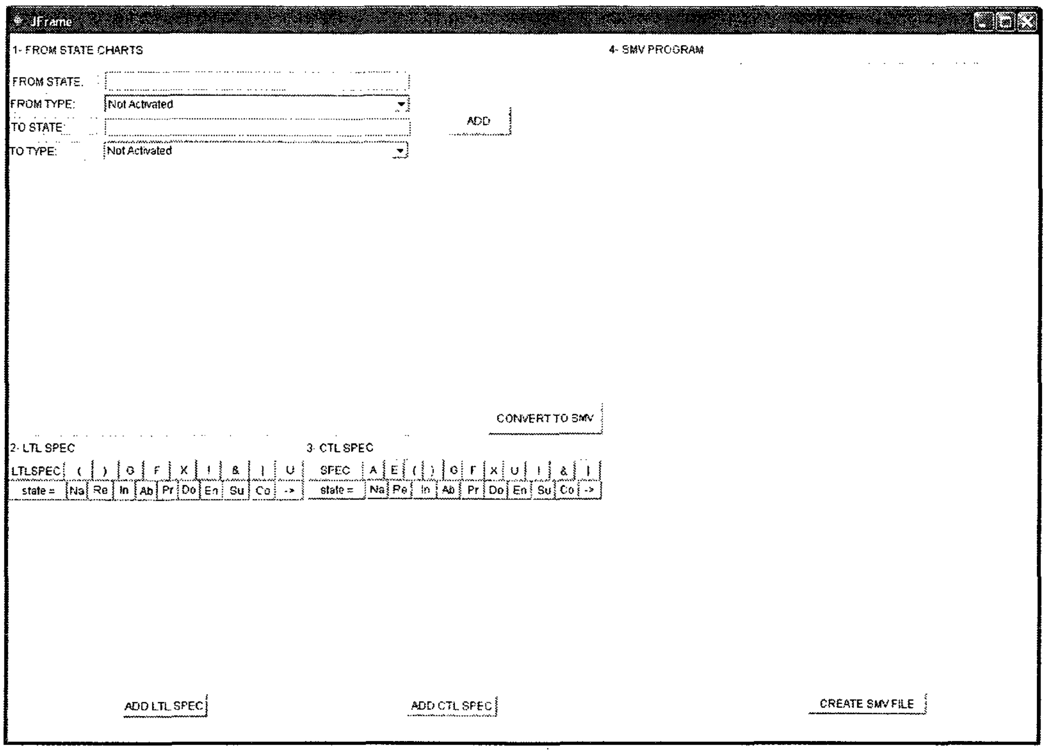

FROM TYPE: Not Activated
TO TYPE: Not Activated

Figure 5.1 SMV Converter Interface

Section 5.2.1 describes the first area and the interaction between the first and fourth area. Section 5.2.2 describes the second and third area as well as their interaction with the fourth area.

5.2.1 From State Charts to SMV

The most important phase is the translation from state charts to SMV syntax.

From the state charts, we will extract the different transitions we have showing the state FROM where the transition is done and the state TO where the transition ends. We assign types to these states to help in the smv translation procedure.

The types we can have are: Not Activated, Receive, Invoke, Processing, Aborted, Done, End, Compensated, Suspended.

Therefore, in this first area, we have 4 fields:

1- FROM STATE field: the user enters in this field the state from where the transition begins

2- FROM TYPE field: the user chooses from the drop-down list of types, the type of the FROM STATE

3- TO STATE field: the user enters in this field the state where the transition ends

4- TO TYPE field: the user chooses from the drop-down list of types, the type of the TO STATE

To facilitate the comprehension of this procedure, we list in a table the different FROM STATE, FROM TYPE, TO STATE and TO TYPE of the ticket reservation system statechart (Figure 2.6) and ATM system statechart. Table 5.1 corresponds to the different transitions of the ticket reservation example. Table 5.2 corresponds to the different transitions of the ATM example.

If we take the first example, we have to enter the information of the table row by row. After entering each row of the table we click on the ADD button.

FROM STATE	FROM TYPE	TO STATE	TO TYPE
INITIAL	NOT ACTIVATED	ITINERARY RECEIVED	RECEIVE
ITINERARY RECEIVED	RECEIVE	AIRLINE INVOKED	INVOKE
AIRLINE INVOKED	INVOKE	AIRLINE RESERVATION COPIED	PROCESSING
AIRLINE INVOKED	INVOKE	AIRLINE RESERVATION CANCELED	ABORTED
AIRLINE RESERVATION COPIED	PROCESSING	HOTEL INVOKED	INVOKE
HOTEL INVOKED	INVOKE	HOTEL RESERVATION COPIED	PROCESSING
HOTEL INVOKED	INVOKE	HOTEL RESERVATION CANCELED	ABORTED
HOTEL RESERVATION COPIED	PROCESSING	VEHICLE INVOKED	INVOKE
VEHICLE INVOKED	INVOKE	VEHICLE RESERVATION COPIED	PROCESSING
VEHICLE INVOKED	INVOKE	VEHICLE RESERVATION CANCELED	ABORTED
VEHICLE RESERVATION COPIED	PROCESSING	ITINERARY MODIFIED	PROCESSING
ITINERARY MODIFIED	PROCESSING	ITINERARY RETURNED	RECEIVE
ITINERARY RETURNED	RECEIVE	DONE	DONE
CANCELED	ABORTED	END	END
DONE	DONE	END	END

Table 5.1 Ticket Reservation System Transitions

The ADD button is responsible to add these information in a table that is not visible to the user. However there is an area below these fields where these information will be shown as transitions (See Figure 5.2).

When all the information are entered, the user clicks on a CONVERT TO SMV button.

This button is responsible of converting all these transitions to the SMV syntax we have in Figure 4.8. The user cannot see what happens backstage.

FROM STATE	FROM TYPE	TO STATE	TO TYPE
INITIAL	NOT ACTIVATED	CARD AND PIN ENTERED	RECEIVE
CARD AND PIN ENTERED	RECEIVE	LOGIN VERIFICATION	INVOKE
LOGIN VERIFICATION	INVOKE	CHOICES	RECEIVE
LOGIN VERIFICATION	INVOKE	CANCELED	ABORTED
CHOICES	RECEIVE	WITHDRAWAL INVOKED	INVOKE
CHOICES	RECEIVE	DEPOSIT INVOKED	INVOKE
CHOICES	RECEIVE	GETBALANCE INVOKED	INVOKE
WITHDRAWAL INVOKED	INVOKE	BALANCE CHECKED	PROCESSING
WITHDRAWAL INVOKED	INVOKE	CANCELED	ABORTED
BALANCE CHECKED	PROCESSING	WITHDRAWAL DONE	RECEIVE
WITHDRAWAL DONE	RECEIVE	RECEIPT INVOKED	INVOKE
WITHDRAWAL DONE	RECEIVE	LOGOUT	DONE
RECEIPT INVOKED	INVOKE	RECEIPT PRINTED	RECEIVE
RECEIPT PRINTED	RECEIVE	LOGOUT	DONE
CANCELED	ABORTED	END	END
LOGOUT	DONE	END	END
DEPOSIT INVOKED	INVOKE	AMOUNT CHECKED	PROCESSING
DEPOSIT INVOKED	INVOKE	CANCELED	ABORTED
AMOUNT CHECKED	PROCESSING	DEPOSIT DONE	RECEIVE
DEPOSIT DONE	RECEIVE	RECEIPT INVOKED	INVOKE
DEPOSIT DONE	RECEIVE	LOGOUT	DONE
GETBALANCE INVOKED	INVOKE	BALANCE DISPLAYED	RECEIVE
BALANCE DISPLAYED	RECEIVE	RECEIPT INVOKED	INVOKE
BALANCE DISPLAYED	RECEIVE	LOGOUT	DONE

Table 5.2 ATM System Transitions

So, when this button is clicked, the first part of the code concerning the transitions will appear in area 4 named: SMV PROGRAM.

A full-example will be shown in Section 1.3

5.2.2 LTL and CTL Specifications

LTL and CTL specifications are extracted from the control behavior (see Section 4.2.2) These specifications will not be extracted automatically in this work. They are defined one time and then they are applied on all operational behaviors.

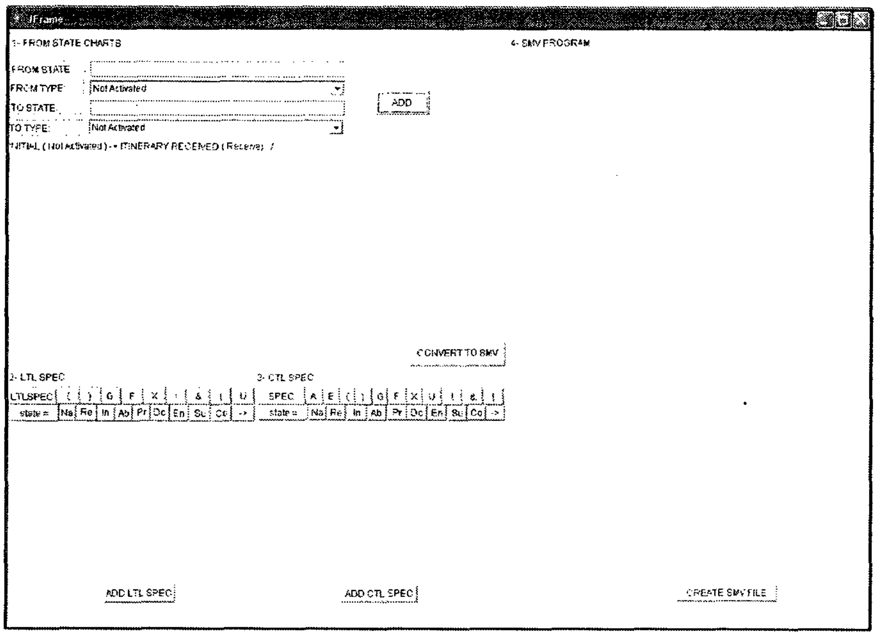
In the second area of the interface of our converter, we have buttons like LTLSPEC, G, F... that helps the user in writing LTL properties in SMV syntax. After the user enters the property or properties, he clicks on the ADD LTL SPEC button available in this area. These specifications will be added to SMV program we are constructing in area 4.

The third area concerns the CTL properties. In this area, we can see buttons that are specific to the CTL syntax like SPEC, A, E... that could be used by the user to write a CTL property in an SMV syntax. Like in the LTL area, after the user enters the property or properties, he clicks on the ADD CTL SPEC button available in this area. These specifications will also be added to SMV program of area 4.

After adding as much properties as he wants, the syntax of the program is shown in area 4. The user will then click on the CREATE SMV FILE button available in the bottom of area 4. Then, the .smv file is created and could be used directly in the NuSMV model checker for verification of the properties, i.e. for verification of the synchronization of messages among web services.

5.3 Step by Step Example

In this section, we will show snapshots of our program, using the ticket reservation system example. We should fill the first area of the interface. Table 5.1 shows all the different transitions we could extract from our ticket reservation system state chart. We can then enter the first row of this table and click on the ADD button. The first transition is subsequently shown below the four fields. (See Figure 5.2)



FROM STATE:
FROM TYPE: Not Activated
TO STATE:
TO TYPE: Not Activated

INITIAL (Not Activated) -> ITINERARY RECEIVED (Receive) /

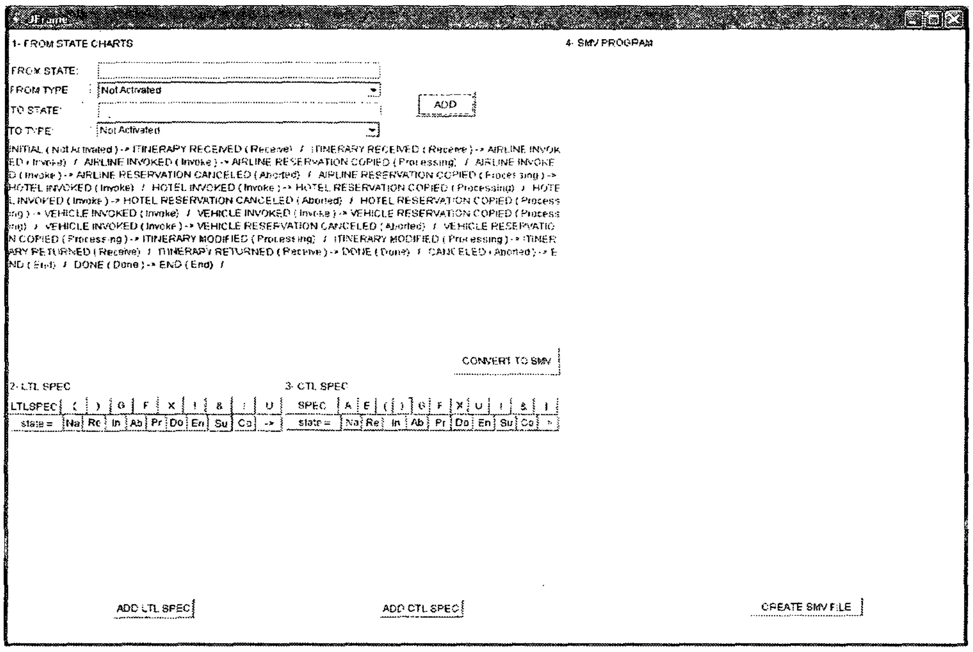
LTLSPEC ( ) G F X ! & | U state = Na Re In Ab Pr Oc En Su Co ->

SPEC A E ( ) G F X U ! & | state = Na Re In Ab Pr Oc En Su Co ->

Figure 5.2 Filling First Box (FROM STATE CHARTS) (1 of 3)

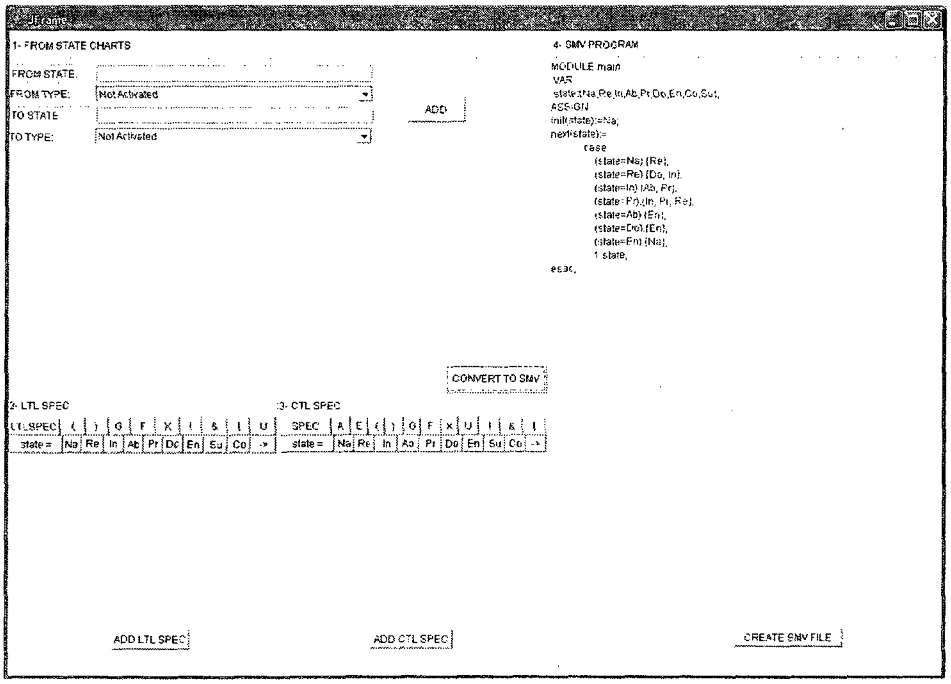
We add all the transitions we could have (from table 5.1) and then we will be in Figure 5.3.

Then, we have to convert this first part to SMV syntax. For that, we click on CONVERT TO SMV button and the syntax obtained will be displayed in the SMV PROGRAM area (see Figure 5.4).



```
INITIAL ( Not Activated ) -> ITINERARY RECEIVED ( Receive)  /  ITINERARY RECEIVED ( Receive ) -> AIRLINE INVOK
ED ( Invoke)  /  AIRLINE INVOKED ( Invoke ) -> AIRLINE RESERVATION COPIED ( Processing)  /  AIRLINE INVOKE
D ( Invoke ) -> AIRLINE RESERVATION CANCELED ( Aborted)  /  AIRLINE RESERVATION COPIED ( Processing ) ->
HOTEL INVOKED ( Invoke)  /  HOTEL INVOKED ( Invoke ) -> HOTEL RESERVATION COPIED ( Processing)  /  HOTE
L INVOKED ( Invoke ) -> HOTEL RESERVATION CANCELED ( Aborted)  /  HOTEL RESERVATION COPIED ( Process
ing ) -> VEHICLE INVOKED ( Invoke)  /  VEHICLE INVOKED ( Invoke ) -> VEHICLE RESERVATION COPIED ( Process
ing)  /  VEHICLE INVOKED ( Invoke ) -> VEHICLE RESERVATION CANCELED ( Aborted)  /  VEHICLE RESERVATIO
N COPIED ( Processing ) -> ITINERARY MODIFIED ( Processing)  /  ITINERARY MODIFIED ( Processing ) -> ITINER
ARY RETURNED ( Receive)  /  ITINERARY RETURNED ( Receive ) -> DONE ( Done)  /  CANCELED ( Aborted ) -> E
ND ( End)  /  DONE ( Done ) -> END ( End)  /
```

Figure 5.3 Filling First Box (FROM STATE CHARTS) (2 of 3)

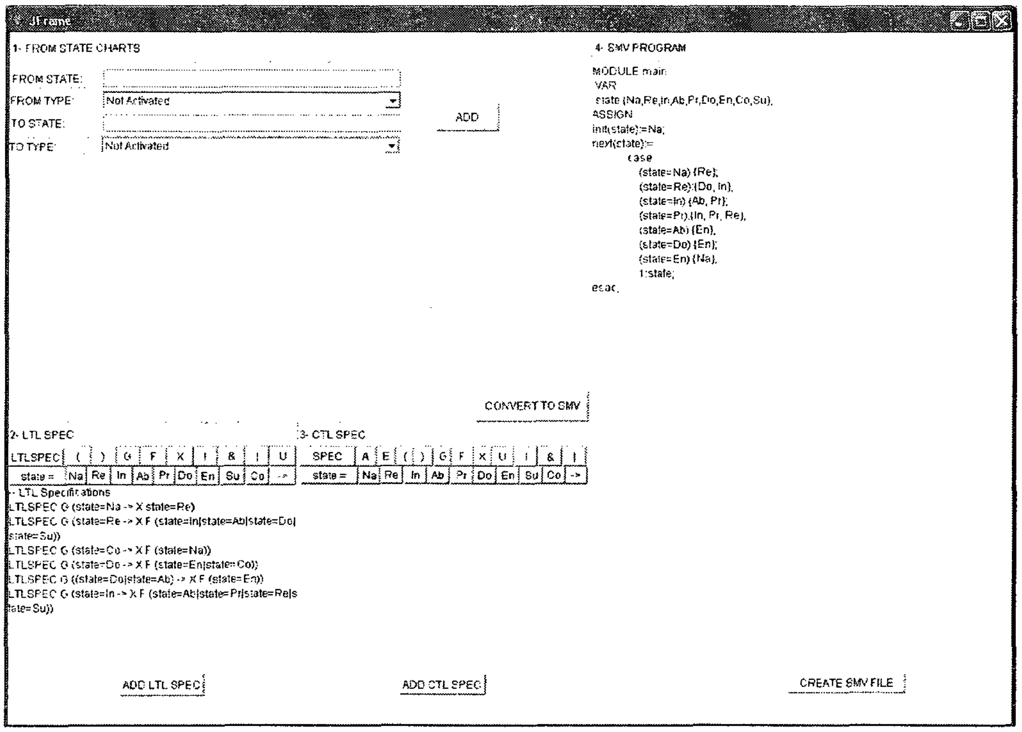


```
MODULE main
 VAR
 state:{Na,Re,In,Ab,Pr,Do,En,Co,Su},
ASSIGN
init(state):=Na;
next(state):=
        case
            (state=Na):{Re},
            (state=Re):{Do, In},
            (state=In):{Ab, Pr},
            (state=Pr):{In, Pr, Re},
            (state=Ab):{En},
            (state=Do):{En},
            (state=En):{Na},
            1:state,
esac,
```

Figure 5.4 Converting First Box to SMV (3 of 3)

After translating the state charts to SMV Syntax, we write in the second area the LTL properties we want to check (Figure 5.5) and we click on ADD LTL SPEC to add them in the SMV program in area 4. Figure 5.6 shows the snapshot after this step. We can also add more LTL specifications (Figures 5.7 and 5.8).

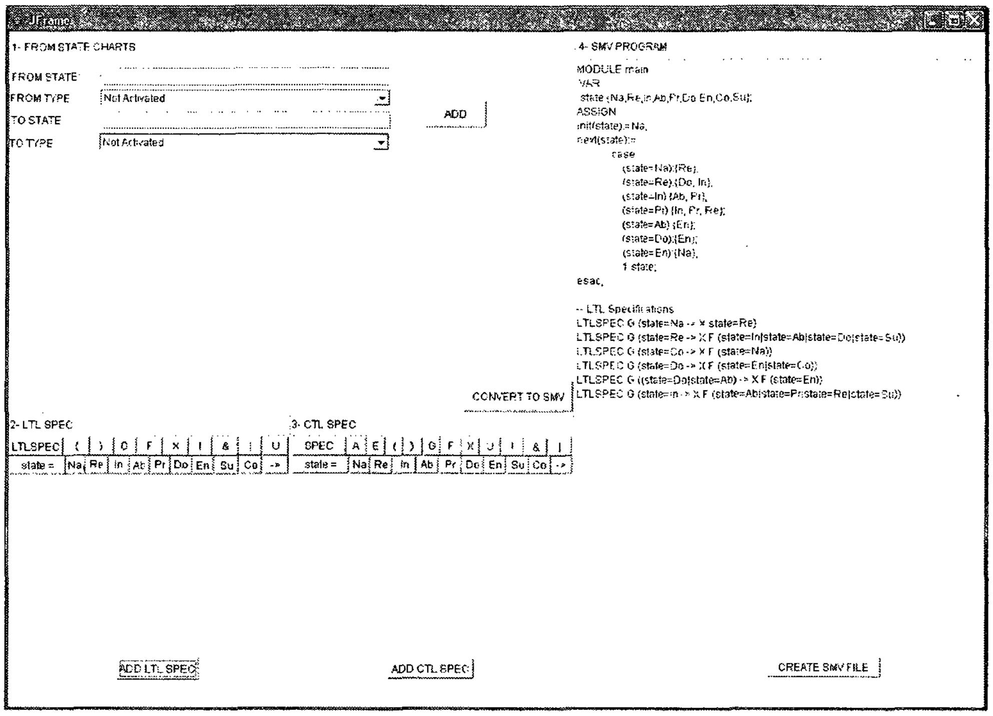
The third area is for CTL properties, so we add these properties in this area and then we click on ADD CTL SPEC to add them to the SMV program (Figures 5.9 and 5.10). As for the LTL specifications, we can also add more CTL specifications in this third field and then append them to the SMV program (Figures 5.11 and 5.12).



```
-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|
state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|s
tate=Su))
```

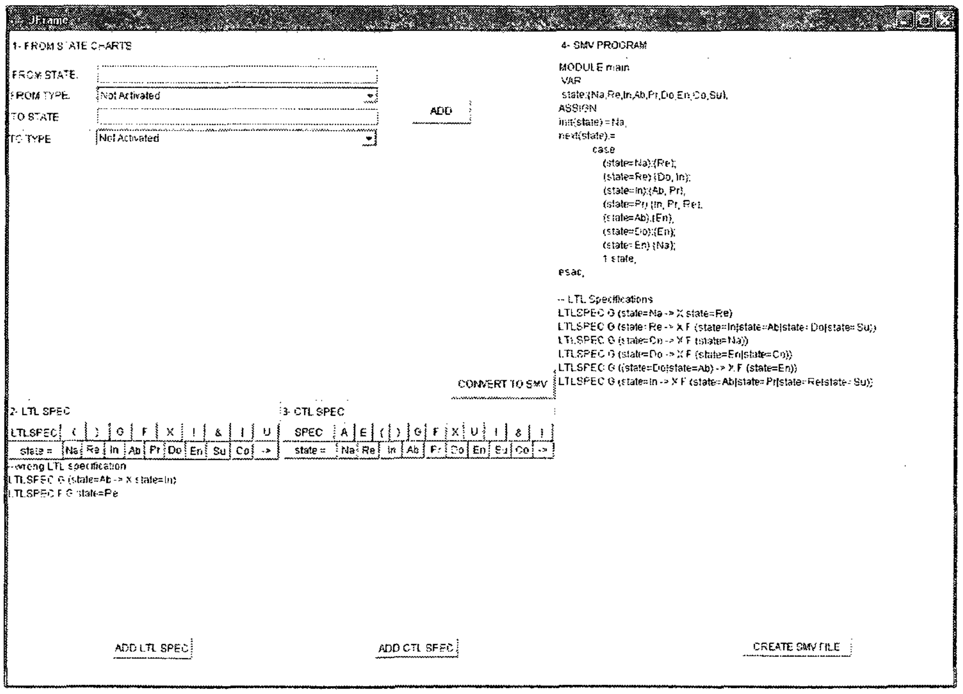
```
MODULE main
 VAR
 state {Na,Re,In,Ab,Pr,Do,En,Co,Su},
ASSIGN
init(state):=Na;
next(state):=
        case
           (state=Na) {Re};
           (state=Re):{Do, In},
           (state=In) {Ab, Pr};
           (state=Pr).{In, Pr, Re},
           (state=Ab) {En},
           (state=Do) {En};
           (state=En) {Na},
           1:state;
esac,
```

Figure 5.5 Filling Second Box (LTL SPEC) (1 of 4)



```
MODULE main
VAR
 state:{Na,Re,In,Ab,Pr,Do,En,Co,Su};
ASSIGN
init(state):=Na;
next(state):=
        case
           (state=Na):{Re};
           (state=Re):{Do, In};
           (state=In):{Ab, Pr};
           (state=Pr):{In, Pr, Re};
           (state=Ab):{En};
           (state=Do):{En};
           (state=En):{Na};
           1:state;
esac;

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))
```

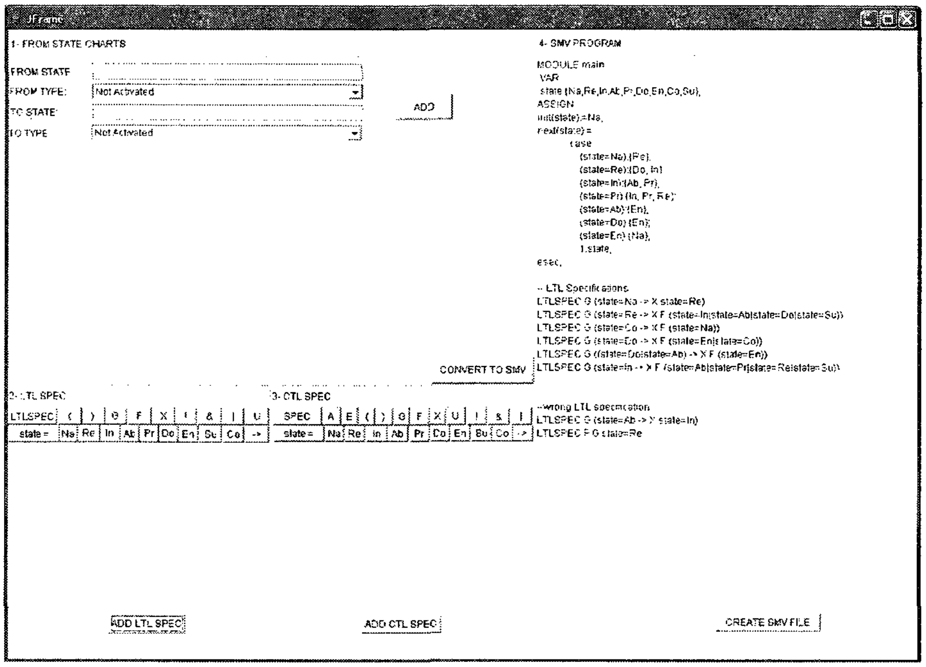
Figure 5.6 Adding LTL SPEC To SMV Program (2 of 4)



```
--wrong LTL specification
LTLSPEC G (state=Ab -> X state=In)
LTLSPEC F G state=Re
```

```
MODULE main
VAR
 state:{Na,Re,In,Ab,Pr,Do,En,Co,Su};
ASSIGN
init(state):=Na;
next(state):=
        case
           (state=Na):{Re};
           (state=Re):{Do, In};
           (state=In):{Ab, Pr};
           (state=Pr):{In, Pr, Re};
           (state=Ab):{En};
           (state=Do):{En};
           (state=En):{Na};
           1:state;
esac;

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))
```

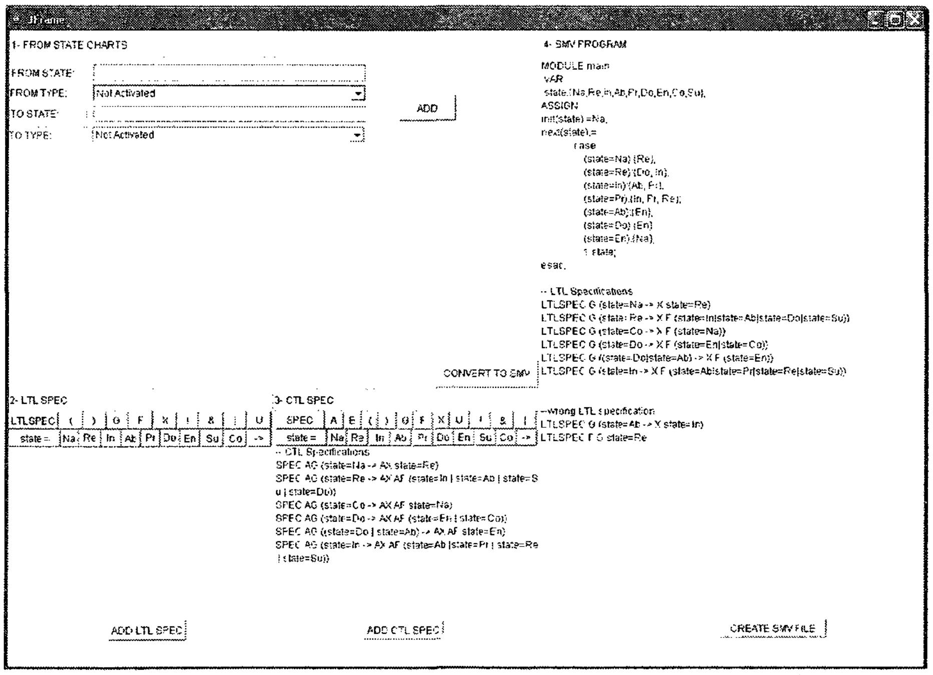
Figure 5.7 Filling Second Box (LTL SPEC) (3 of 4)


JFrame
1- FROM STATE CHARTS
ADD
CONVERT TO SMV
2- LTL SPEC
LTLSPEC ( ) G F X ! & | U
state = Na Re In Ab Pr Do En Su Co ->
3- CTL SPEC
SPEC A E ( ) G F X U ! & |
state = Na Re In Ab Pr Do En Su Co ->
ADD LTL SPEC
ADD CTL SPEC
CREATE SMV FILE
4- SMV PROGRAM

FROM STATE:
FROM TYPE: Not Activated
TO STATE:
TO TYPE: Not Activated

```
MODULE main
 VAR
 state {Na,Re,In,Ab,Pr,Do,En,Co,Su},
ASSIGN
init(state):=Na,
next(state)=
        case
           (state=Na):{Re},
           (state=Re):{Do, In}
           (state=In):{Ab, Pr},
           (state=Pr):{In, Pr, Re};
           (state=Ab):{En},
           (state=Do):{En};
           (state=En):{Na},
           1:state,
esac,

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))

--wrong LTL specification
LTLSPEC G (state=Ab -> X state=In)
LTLSPEC F G state=Re
```

Figure 5.8 Adding LTL SPEC To SMV Program (4 of 4)

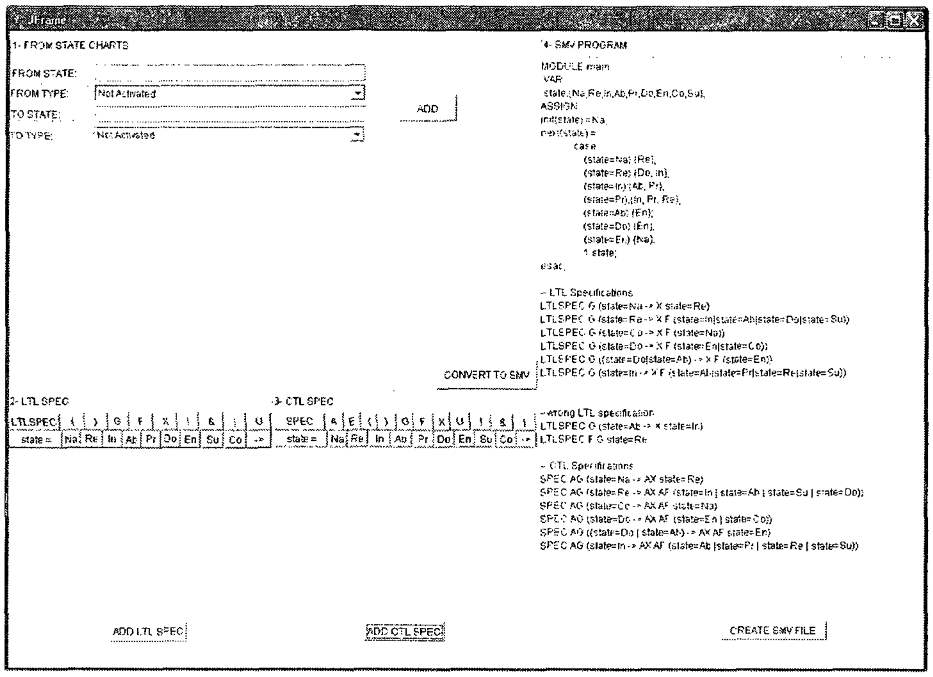

JFrame
1- FROM STATE CHARTS
ADD
CONVERT TO SMV
2- LTL SPEC
LTLSPEC ( ) G F X ! & | U
state = Na Re In Ab Pr Do En Su Co ->
3- CTL SPEC
SPEC A E ( ) G F X U ! & |
state = Na Re In Ab Pr Do En Su Co ->
ADD LTL SPEC
ADD CTL SPEC
CREATE SMV FILE
4- SMV PROGRAM

FROM STATE:
FROM TYPE: Not Activated
TO STATE:
TO TYPE: Not Activated

```
-- CTL Specifications
SPEC AG (state=Na -> AX state=Re)
SPEC AG (state=Re -> AX AF (state=In | state=Ab | state=S
u | state=Do))
SPEC AG (state=Co -> AX AF state=Na)
SPEC AG (state=Do -> AX AF (state=En | state=Co))
SPEC AG ((state=Do | state=Ab) -> AX AF state=En)
SPEC AG (state=In -> AX AF (state=Ab |state=Pr | state=Re
 | state=Su))
```

```
MODULE main
 VAR
 state:{Na,Re,In,Ab,Pr,Do,En,Co,Su},
ASSIGN
init(state) =Na,
next(state):=
        case
           (state=Na):{Re},
           (state=Re):{Do, In},
           (state=In):{Ab, Pr},
           (state=Pr):{In, Pr, Re};
           (state=Ab):{En},
           (state=Do):{En}
           (state=En):{Na},
           1:state;
esac,

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))

--wrong LTL specification
LTLSPEC G (state=Ab -> X state=In)
LTLSPEC F G state=Re
```

Figure 5.9 Filling Third Box (CTL SPEC) (1 of 4)

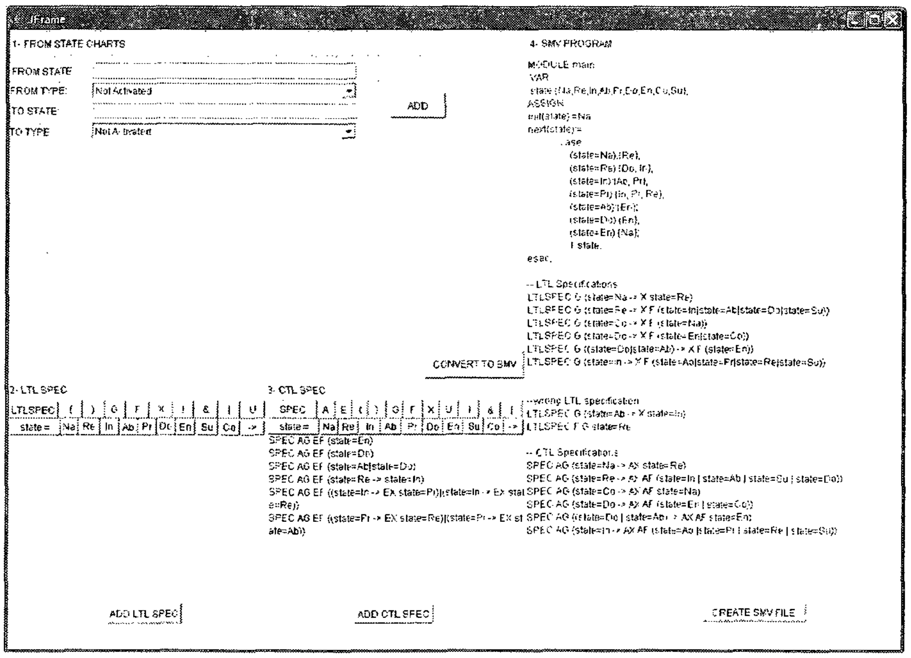

JFrame
1- FROM STATE CHARTS
ADD
CONVERT TO SMV
2- LTL SPEC
LTLSPEC ( ) G F X ! & | U
state = Na Re In Ab Pr Do En Su Co ->
3- CTL SPEC
SPEC A E ( ) G F X U ! & |
state = Na Re In Ab Pr Do En Su Co ->
ADD LTL SPEC
ADD CTL SPEC
CREATE SMV FILE

FROM STATE:
FROM TYPE: Not Activated
TO STATE:
TO TYPE: Not Activated

4- SMV PROGRAM

```
MODULE main
 VAR
 state:{Na,Re,In,Ab,Pr,Do,En,Co,Su};
ASSIGN
init(state) =Na;
next(state) =
        case
          (state=Na):{Re};
          (state=Re):{Do, In};
          (state=In):{Ab, Pr};
          (state=Pr):{In, Pr, Re};
          (state=Ab):{En};
          (state=Do):{En};
          (state=En):{Na};
          1: state;
esac;

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))

--wrong LTL specification
LTLSPEC G (state=Ab -> X state=In)
LTLSPEC F G state=Re

-- CTL Specifications
SPEC AG (state=Na -> AX state=Re)
SPEC AG (state=Re -> AX AF (state=In | state=Ab | state=Su | state=Do));
SPEC AG (state=Co -> AX AF state=Na)
SPEC AG (state=Do -> AX AF (state=En | state=Co))
SPEC AG ((state=Do | state=Ab) -> AX AF state=En)
SPEC AG (state=In -> AX AF (state=Ab |state=Pr | state=Re | state=Su))
```

Figure 5.10 Adding CTL SPEC To SMV Program (2 of 4)


JFrame
1- FROM STATE CHARTS
ADD
CONVERT TO SMV
2- LTL SPEC
LTLSPEC ( ) G F X ! & | U
state = Na Re In Ab Pr Do En Su Co ->
3- CTL SPEC
SPEC A E ( ) G F X U ! & |
state = Na Re In Ab Pr Do En Su Co ->
ADD LTL SPEC
ADD CTL SPEC
CREATE SMV FILE

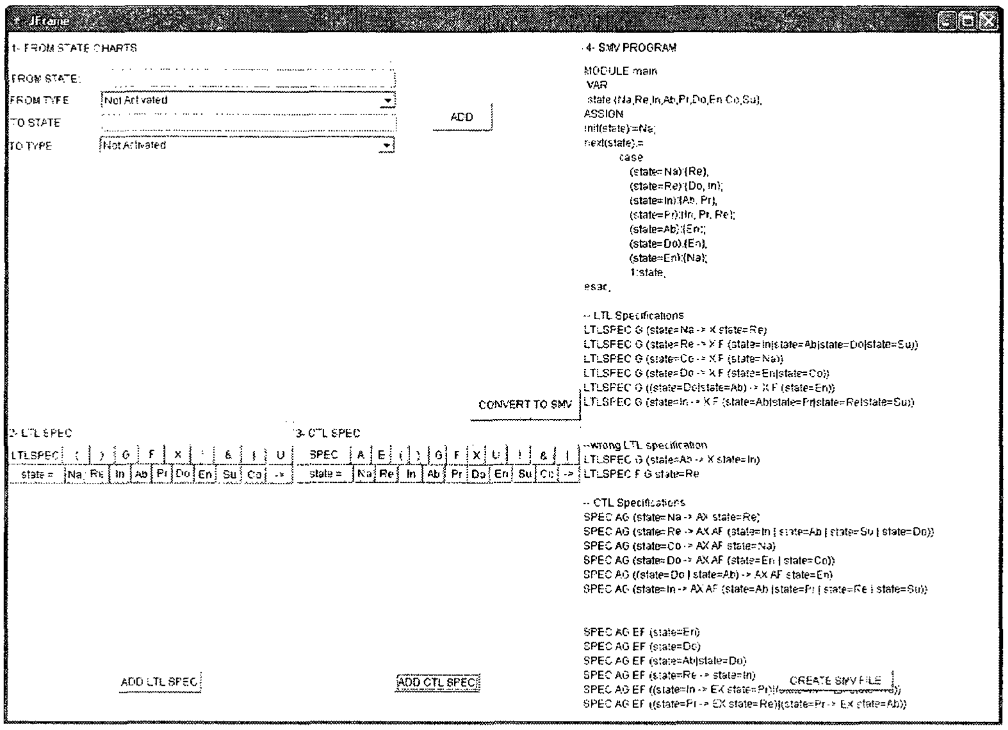
FROM STATE:
FROM TYPE: Not Activated
TO STATE:
TO TYPE: Not Activated

```
SPEC AG EF (state=En)
SPEC AG EF (state=Do)
SPEC AG EF (state=Ab|state=Do)
SPEC AG EF (state=Re -> state=In)
SPEC AG EF ((state=In -> EX state=Pr)|(state=In -> EX stat
e=Re))
SPEC AG EF ((state=Pr -> EX state=Re)|(state=Pr -> EX st
ate=Ab))
```

4- SMV PROGRAM

```
MODULE main
 VAR
 state:{Na,Re,In,Ab,Pr,Do,En,Co,Su};
ASSIGN
init(state) =Na;
next(state)=
        case
          (state=Na):{Re};
          (state=Re):{Do, In};
          (state=In):{Ab, Pr};
          (state=Pr):{In, Pr, Re};
          (state=Ab):{En};
          (state=Do):{En};
          (state=En):{Na};
          1: state;
esac;

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))

--wrong LTL specification
LTLSPEC G (state=Ab -> X state=In)
LTLSPEC F G state=Re

-- CTL Specifications
SPEC AG (state=Na -> AX state=Re)
SPEC AG (state=Re -> AX AF (state=In | state=Ab | state=Su | state=Do))
SPEC AG (state=Co -> AX AF state=Na)
SPEC AG (state=Do -> AX AF (state=En | state=Co))
SPEC AG ((state=Do | state=Ab) -> AX AF state=En)
SPEC AG (state=In -> AX AF (state=Ab |state=Pr | state=Re | state=Su))
```

Figure 5.11 Filling Third Box (CTL SPEC) (3 of 4)



```
MODULE main
 VAR
 state {Na,Re,In,Ab,Pr,Do,En Co,Su},
ASSIGN
init(state):=Na;
next(state):=
       case
          (state=Na):{Re},
          (state=Re):{Do, In};
          (state=In):{Ab, Pr},
          (state=Pr):{In, Pr, Re};
          (state=Ab):{En};
          (state=Do):{En},
          (state=En):{Na};
          1:state,
esac,

-- LTL Specifications
LTLSPEC G (state=Na -> X state=Re)
LTLSPEC G (state=Re -> X F (state=In|state=Ab|state=Do|state=Su))
LTLSPEC G (state=Co -> X F (state=Na))
LTLSPEC G (state=Do -> X F (state=En|state=Co))
LTLSPEC G ((state=Do|state=Ab) -> X F (state=En))
LTLSPEC G (state=In -> X F (state=Ab|state=Pr|state=Re|state=Su))


--wrong LTL specification
LTLSPEC G (state=Ab -> X state=In)
LTLSPEC F G state=Re

-- CTL Specifications
SPEC AG (state=Na -> AX state=Re)
SPEC AG (state=Re -> AX AF (state=In | state=Ab | state=Su | state=Do))
SPEC AG (state=Co -> AX AF state=Na)
SPEC AG (state=Do -> AX AF (state=En | state=Co))
SPEC AG ((state=Do | state=Ab) -> AX AF state=En)
SPEC AG (state=In -> AX AF (state=Ab |state=Pr | state=Re | state=Su))


SPEC AG EF (state=En)
SPEC AG EF (state=Do)
SPEC AG EF (state=Ab|state=Do)
SPEC AG EF (state=Re -> state=In)
SPEC AG EF ((state=In -> EX state=Pr)|[illegible]e))
SPEC AG EF ((state=Pr -> EX state=Re)|(state=Pr -> EX state=Ab))
```

Figure 5.12 Adding CTL SPEC To SMV Program (4 of 4)

Chapter 6: Conclusions and Future Work

6.1 Conclusion

In this dissertation, we presented a formal verification approach of conversations in composite web services. In Chapters 2 and 3, we gave an overview of composition and model checking concepts. We then proposed an approach for modeling composite web services based on two behaviors: control and operational. The operational behavior shows the business logic of the process functionality for a composite web service. The control behavior shows the constraints and states that the operational behavior should be in. These two behaviors are linked together to check the synchronization between the conversations of composite services. We use state charts enhanced with additional syntax to facilitate the mapping process between the two behaviors. Synchronizing both behaviors is a key issue in designing good conversations between different web services that participate in composite services. We used symbolic model checking as the verification approach. The properties to be checked are taken from the control behavior and verified in the different operational scenarios to check the correctness of conversations among web services.

Our main contribution is the formal and automatic verification of the mapping procedure using symbolic model checking technique. A second contribution is the creation of a Java-based translation procedure which in addition to the NuSMV model checker contributes to the implementation of our verification model.

6.2 Future Work

In this thesis, we only considered centralized processes and orchestration in composition. As future work, we plan to extend this approach for choreography-based composition. Taking a choreography composition that does not have any controller process is a challenging issue. In choreography, all the participating web services know the actual business process and are well aware of which web services they need to interact with and when to execute the operations. Consequently, we need a control behavior that corresponds to a choreography process, which is very dynamic.

Also, fault handling is easier in orchestration as the execution is controlled, which is not the case with choreography. Web services can be easily and transparently replaced in case of orchestration as the involved web services do not know the actual business process, whereas it will be difficult in case of choreography.

Last but not least, we plan to verify other types of conversation between web services such as negotiation and argumentation which are used in other web services applications, for instance communities of web services.

References

[1] D. Amyot, T. Gray, R. Liscano, L. Logrippo, and J. Sincennes. Interactive conflict detection and resolution for personalized services. *Journal of Communication Networks* 7:353-66, 2005

[2] H. Baumeister. System Integration. Informatics and Mathematical Modeling. *Technical University of Denmark.* Spring 2008

[3] D. Benslimane, Z. Maamar, C. Ghedira. How to Track Composite Web Services? A Solution Based On the Concept Of View. *Journal of Electronic Commerce Research*, 7(3), 2006

[4] J. Bentahar. Formal Methods for Software. *Quality Methodologies for Software (INSE 6250/4-UU).* Winter 2008. Principles of Model Checking by Joost-Pieter Katoen.

[5] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard. Web Services Architecture. W3C Working Group Note http://www.w3.org/TR/ws-arch/, 11 February 2004

[6] T. Bultan, J. Su and X. Fu. Analyzing Conversations of Web Services. *IEEE Internet Computing*, 10(1):16-25. February 2006

[7] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. *In Proceeding of International Conference on Computer-Aided Verification (CAV 2002).* Copenhagen, Denmark, July 27-31, 2002

[8] E. M. Clarke and E. A. Emerson. A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications, *ACM Transactions on Programming Languages and Systems* 8: 244,doi:10.1145/5397.5399, 1986

[9] E. M. Clarke and E. A. Emerson. Characterizing correctness properties of parallel programs using fixpoints, *Automata, Languages and Programming*, doi:10.1007/3-540-10003-2_69, 1980.

[10] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Logic of Programs* 1981: 52-71.

[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. (2000), Counterexample-Guided Abstraction Refinement, *Computer Aided Verification* 1855: 154, doi:10.1007/10722167_15

[12] E.M. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, 1999

[13] Y. Ding and Y. Zhang. System Modification Case Studies. *IEEE computer society*. 2007

[14] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of Web service compositions. *In Proc. the 18th IEEE Int. Conf. on Automated Software Engineering Conference (ASE),* 2003

[15] X. Fu, T. Bultan and J.Su. Analysis of Interacting BPEL Web Services. *In Proc. Int. World Wide Web Conf. (WWW)*, 2004

[16] X. Fu, T. Bultan, and J. Su. Model Checking Interactions of Composite Web Services. *University of California at Santa Barbara*. 2004

[17] H. S. Hong, I. Lee and O. Sokolsky. Automatic Test Generation from Statecharts Using Model Checking. *University of Pennsylvania*. 2001

[18] http://archive.devx.com/javasr/articles/gabhart/gabhart-1.asp

[19] http://www.cs.cmu.edu/~emc/15817-s05/smv.ppt

[20] http://www.ibm.com/developerworks/library/w-ovr/ Web Services architecture overview. The next stage of evolution for e-business IBM Services Architecture Team, Writers, IBM, Software Group

[21] http://www.w3.org/TR/ws-gloss. Web Services Architecture, February 2003

[22] R. Hull and J. Su. Tools for Composite Web Services: A Short Overview Source. *SIGMOD Record*, 34(2):86–95. 2005

[23] R. Khalaf, N. Mukhi, and S.Weerawarana. *Service Oriented Composition in BPEL4WS.* In Proc. WWW'03, 2003.

[24] A. Layouni, L. Logrippo, and K. J. Turner. Conflict Detection in Call Control Using First-Order Logic Model Checking. *In: L. DuBousquet, Jean-Luc Richier (Eds): 9th International Conference on Feature Interactions in Software and Communication Systems*, 77-92, IOS Press, 2008

[25] F. Leymann andW. Altenhuber. Managing Business ... Web Services Flow Language (WSFL. 1.0). *IBM Corporation*, May 2001.

[26] Z. Maamar, D. Benslimane, C. Cherida and M. Mrissa. Views in Composite Web Services. *IEEE internet computing*, 79-84, August 2005

[27] Z. Maamar, Q. Sheng, H. Yahyaoui, J.Bentahar and K. Boukadi. A New Approach to Model Web Services' Behaviors based on Synchronization. *Fifth International Symposium on Frontiers of Information Systems and Network Applications (FINA'2009), Bradford, UK,* 2009

[28] K. L. McMillan, Kluwer. Symbolic Model Checking, ISBN 0-7923-9380-5

[29] S. Meng and F. Arbab. Web Services Choreography and Orchestration in Reo and Constraint Automata. *CWI, Amsterdam, The Netherlands*

[30] C. Peltz. Web Services Orchestration and Choreography, Computer, 36(10):46-52, Oct. 2003

[31] C. Peltz. Web Services Orchestration: A Review of Emerging Technologies, Tools, and Standards. *Hewlett Packard, Co*, January 2003

[32] ACM Turing Award Honors Founders of Automatic Verification Technology, *Press Release*

[33] J. P. Queille, J. Sifakis. Specification and verification of concurrent systems in CESAR, *International Symposium on Programming*, doi:10.1007/3-540-11494-7_22, 1982.

[34] S. Robak and B. Franczyk. Modeling Web Services Variability with Feature Diagrams. In *Web, Web-Services, and Database Systems*, pages 120-128, 2002.

[35] J. Su, T. Bultan and X. Fu. Web Service Interactions: Analysis and Design. *Proceedings of the Second International Workshop on Semantic and Dynamic Web Processes (SDWP 2005)*, pp. 14-19, Orlando, Florida, USA, 2005

[36] *USACM*: 2007 Turing Award Winners Announced

[37] M. Weiss and B. Esfandiari. On Feature Interactions among Web Services. *International Journal of Web Services Research*, 2(4), 21-45, 2005

[38] S. XLANG. Web Services for Business Process Design, *Microsoft*, 2001

[39] H. Yahyaoui, Z. Maamar and K. Boukadi. Web Services Synchronization in Composition Scenarios: The Centralized View. *The International Conference on Information Science, Technology and Applications (ISTA 2009), Kuwait,* 2009

Appendices

Appendix 1: SMV Converter Source Code

```
private Button getButton3() {
        if (button3 == null) {
            button3 = new Button();
            button3.setBounds(new java.awt.Rectangle(123,710,92,25));
            button3.setLabel("ADD LTL SPEC");
            button3.addActionListener(
                    new ActionListener() {


                          public void actionPerformed( ActionEvent event )
                            {
                          jTextArea4.append(jTextArea2.getText() + "\n");
                                   jTextArea2.setText("");

                            }

                          } // end anonymous inner class

                        ); // end call to addActionListener
            }
        return button3;
    }
/**
    * This method initializes buttonCTL
    *
    * @return java.awt.Button
    */
    private Button getButtonCTL() {
        if (buttonCTL == null) {
            buttonCTL = new Button();
            buttonCTL.setBounds(new java.awt.Rectangle(429,711,96,23));
            buttonCTL.setLabel("ADD CTL SPEC");
            buttonCTL.addActionListener(
                    new ActionListener() {


            public void actionPerformed( ActionEvent event )
                            {
                  jTextArea4.append("\n"+ jTextArea3.getText() + "\n");
                                jTextArea3.setText("");
```

```
                                    }

                                  }  // end anonymous inner class

                                ); // end call to addActionListener
                }
                return buttonCTL;
        }
/**
         * This method initializes buttonSMV
         *
         * @return java.awt.Button
         */
        private Button getButtonSMV() {
                if (buttonSMV == null) {
                        buttonSMV = new Button();
                        buttonSMV.setBounds(new java.awt.Rectangle(857,709,128,23));
                        buttonSMV.setLabel("CREATE SMV FILE");
                        buttonSMV.addActionListener(
                                new ActionListener() {


                              public void actionPerformed( ActionEvent event )
                                {
                                 try {
                                   String lines[] = jTextArea4.getText().split("\\n");

                                 FileWriter ryt=new FileWriter("C:\\Program
                                 Files\\NuSMV\\2.4.3\\ smvprogram.smv");


                                     for(int i = 0; i < lines.length; i++) {

                                                 ryt.write(lines[i]);
                                                 ryt.write("\r\n");
                                                 ryt.flush();

                                          }

                                   } catch(IOException e) {
                                           e.printStackTrace();
                                          }
                                   jTextArea4.setText("");

                                   }

                                  }  // end anonymous inner class

                                ); // end call to addActionListener
                }
                return buttonSMV;
        }
/**
         * This method initializes buttonDB
         *
         * @return java.awt.Button
         */
        private Button getButtonDB() {
                if (buttonDB == null) {
                        buttonDB = new Button();
                        buttonDB.setBounds(new java.awt.Rectangle(470,78,68,31));
                        buttonDB.setLabel("ADD");


                        buttonDB.addActionListener(
                                new ActionListener() {


                                        public void actionPerformed( ActionEvent
event )
                                        {
jTextArea1.append(jTextFieldFROM.getText() + " ( " + choice1.getSelectedItem() + " ) -> " +
jTextFieldTO.getText()+ " ( " + choice2.getSelectedItem()+ ")  /   ");

                                         String firstchoice="";
                                         String secondchoice="";
                                         //for first choice
                                        if(choice1.getSelectedItem()=="Not Activated")
                                                firstchoice = "Na";
                                         if(choice1.getSelectedItem()=="Receive")
                                                firstchoice = "Re";
                                         if(choice1.getSelectedItem()=="Invoke")
                                                firstchoice = "In";
                                         if(choice1.getSelectedItem()=="Processing")
                                                firstchoice = "Pr";
                                         if(choice1.getSelectedItem()=="Aborted")
                                                firstchoice = "Ab";
                                         if(choice1.getSelectedItem()=="Done")
                                                firstchoice = "Do";
                                         if(choice1.getSelectedItem()=="End")
                                                firstchoice = "En";

                                        if(choice1.getSelectedItem()=="Compensated")
                                                firstchoice = "Co";
                                         if(choice1.getSelectedItem()=="Suspended")
                                                firstchoice = "Su";

                                         // for second choice
                                        if(choice2.getSelectedItem()=="Not Activated")
                                                secondchoice = "Na";
                                        if(choice2.getSelectedItem()=="Receive")
                                                secondchoice = "Re";
                                        if(choice2.getSelectedItem()=="Invoke")
                                                secondchoice = "In";
                                        if(choice2.getSelectedItem()=="Processing")
                                                secondchoice = "Pr";
                                        if(choice2.getSelectedItem()=="Aborted")
                                                secondchoice = "Ab";
                                        if(choice2.getSelectedItem()=="Done")
                                                secondchoice = "Do";
                                        if(choice2.getSelectedItem()=="End")
                                                secondchoice = "En";

                                    if(choice2.getSelectedItem()=="Compensated")
                                                secondchoice = "Co";
                                        if(choice2.getSelectedItem()=="Suspended")
                                                secondchoice = "Su";


                                        try
                                        {
                                        Statement stmt;
                                        stmt = connection.createStatement();
                                String insertString ="INSERT INTO dba(
                    FROM_STATE, FROM_TYPE, TO_STATE, TO_TYPE )" +
                    "VALUES ('"+ jTextFieldFROM.getText()+"', '"+ firstchoice + "', '"
                    +jTextFieldTO.getText() + "', '"+ secondchoice +"')";
                                int counting = stmt.executeUpdate(insertString);
                                stmt.close();

                                        }
                                    catch ( SQLException sqlex ) {
                                                sqlex.printStackTrace();
                                            }

                                        jTextFieldFROM.setText("");
                                          jTextFieldTO.setText("");
                                                choice1.select("Not Activated");
                                                choice2.select("Not Activated");
                                        }

                                  } // end anonymous inner class

                                ); // end call to addActionListener
            }
            return buttonDB;
    }
```

```
/**
         * This method initializes buttonCONVERT
         *
         * @return java.awt.Button
         */
        private Button getButtonCONVERT() {
                if (buttonCONVERT == null) {
                        buttonCONVERT = new Button();
                        buttonCONVERT.setBounds(new java.awt.Rectangle(514,396,123,35));
                        buttonCONVERT.setLabel("CONVERT TO SMV");
                        buttonCONVERT.addActionListener(
                                        new ActionListener() {


                                          public void actionPerformed( ActionEvent event )
                                                  {
                                        jTextArea4.append("MODULE main \n VAR \n " +

                                                  "state:{Na,Re,In,Ab,Pr,Do,En,Co,Su};\n" +
                                                   "ASSIGN\n" + "init(state):=Na;\n" +
                                                   "next(state):= \n"+
                                                          "          case\n");
                                                          jTextArea1.setText("");


                                                    try {
                                                    Statement statement;
      //----------------------NOTACTIVATED--------------------------------------------

                ResultSet rsNa;
                String outputNa="";
                statement = connection.createStatement();
                ResultSet resultSetNa = statement.executeQuery("SELECT COUNT(*) FROM"
                +
                " (SELECT DISTINCT TO_TYPE FROM dba WHERE FROM_TYPE='Na')");

                // Get the number of rows from the result set
                resultSetNa.next();
                int NrowsNa = resultSetNa.getInt(1);
                resultSetNa.close();
                statement.close();

                if (NrowsNa>=1)
                {
                jTextArea4.append("            (state=Na):{");
                statement = connection.createStatement();
                rsNa = statement.executeQuery( "SELECT DISTINCT TO_TYPE " +
                "FROM dba WHERE FROM_TYPE='Na'" );
                while(rsNa.next())
                {
                if(rsNa.getRow() < NrowsNa)
                outputNa += rsNa.getString("TO_TYPE")+ ", ";
                else
                  outputNa += rsNa.getString("TO_TYPE");
                       }
                rsNa.close();
                statement.close();
                jTextArea4.append(outputNa+"};\n");
                }

//----------------------RECEIVE----------------------------------------------------
    ResultSet rsRe;
    String outputRe="";
     statement = connection.createStatement();
    ResultSet resultSetRe = statement.executeQuery("SELECT COUNT(*) FROM" +
    " (SELECT DISTINCT TO_TYPE FROM dba WHERE FROM_TYPE='Re')");

    // Get the number of rows from the result set
                          resultSetRe.next();
                                   int NrowsRe = resultSetRe.getInt(1);
                                              resultSetRe.close();
                                               statement.close();

                  if (NrowsRe>=1)
                                    {
                    jTextArea4.append("              (state=Re): {");
                  statement = connection.createStatement();
                  rsRe = statement.executeQuery( "SELECT DISTINCT TO_TYPE " +
                  "FROM dba WHERE FROM_TYPE='Re'" );
                  while(rsRe.next())
                                    {
                                    if(rsRe.getRow() < NrowsRe)
                                    outputRe += rsRe.getString("TO_TYPE")+ ", ";
                                    else
                                    outputRe += rsRe.getString("TO_TYPE");
                                                         }
                                    rsRe.close();
                                    statement.close();

                  jTextArea4.append(outputRe+"};\n");
                                                         }


                //----------------------INVOKE---------------------------------------------------
                ResultSet rsIn;
                String outputIn="";
                statement = connection.createStatement();
                ResultSet resultSetIn = statement.executeQuery("SELECT COUNT(*) FROM" +
                                " (SELECT DISTINCT TO_TYPE FROM dba WHERE
                                FROM_TYPE='In')");
```

```
        // Get the number of rows from the result set
         resultSetIn.next();
        int NrowsIn = resultSetIn.getInt(1);
        resultSetIn.close();
        statement.close();

        if (NrowsIn>=1)
        {
        jTextArea4.append("                (state=In): {");
         statement = connection.createStatement();
        rsIn = statement.executeQuery( "SELECT DISTINCT TO_TYPE " +
        "FROM dba WHERE FROM_TYPE='In'" );
        while(rsIn.next())
                {
        if(rsIn.getRow() < NrowsIn)
          outputIn += rsIn.getString("TO_TYPE")+ ", ";
                else
          outputIn += rsIn.getString("TO_TYPE");
                                    }
        rsIn.close();
        statement.close();
        jTextArea4.append(outputIn+"};\n");
        }
//-----------------------Processing---------------------------------------------------
        ResultSet rsPr;
        String outputPr="";
        statement = connection.createStatement();
        ResultSet resultSetPr = statement.executeQuery("SELECT COUNT(*) FROM"
        +" (SELECT DISTINCT TO_TYPE FROM dba WHERE FROM_TYPE='Pr')");

        // Get the number of rows from the result set

        resultSetPr.next();
        int NrowsPr = resultSetPr.getInt(1);
        resultSetPr.close();
        statement.close();

        if (NrowsPr>=1)
        {       jTextArea4.append("                (state=Pr): {");
                statement = connection.createStatement();
                rsPr = statement.executeQuery( "SELECT DISTINCT TO_TYPE " +
                "FROM dba WHERE FROM_TYPE='Pr'" );

                while(rsPr.next())
                {

                if(rsPr.getRow() < NrowsPr)
                outputPr += rsPr.getString("TO_TYPE")+ ", ";
                else
                outputPr += rsPr.getString("TO_TYPE");
                                                                }
                rsPr.close();
                statement.close();

                jTextArea4.append(outputPr+"};\n");
        }
                                                                                      //--
--------------------ABORTED---------------------------------------------------

ResultSet rsAb;

String outputAb="";

statement = connection.createStatement();

ResultSet resultSetAb = statement.executeQuery("SELECT COUNT(*) FROM" +

                " (SELECT DISTINCT TO_TYPE FROM dba WHERE FROM_TYPE='Ab')");


// Get the number of rows from the result set

resultSetAb.next();

int NrowsAb = resultSetAb.getInt(1);

resultSetAb.close();

statement.close();


if (NrowsAb>=1)

{

jTextArea4.append("            (state=Ab):{");

statement = connection.createStatement();

rsAb = statement.executeQuery( "SELECT DISTINCT TO_TYPE " +

                "FROM dba WHERE FROM_TYPE='Ab'" );

while(rsAb.next())

{

        if(rsAb.getRow() < NrowsAb)

        outputAb += rsAb.getString("TO_TYPE")+ ", ";

        else
```

```
            outputAb += rsAb.getString("TO_TYPE");

        }

rsAb.close();

statement.close();

jTextArea4.append(outputAb+"};\n");

}

                //----------------------DONE-----------------------------------------------------------

                  ResultSet rsDo;

                  String outputDo="";

                  statement = connection.createStatement();
                   ResultSet resultSetDo = statement.executeQuery("SELECT COUNT(*)
                   FROM" +" (SELECT DISTINCT TO_TYPE FROM dba WHERE
                   FROM_TYPE='Do')");


                    // Get the number of rows from the result set

                        resultSetDo.next();

                        int NrowsDo = resultSetDo.getInt(1);

                           resultSetDo.close();

                           statement.close();


                           if (NrowsDo>=1)

                           {

                             jTextArea4.append("              (state=Do):{");

                           statement = connection.createStatement();

                         rsDo = statement.executeQuery( "SELECT DISTINCT TO_TYPE " +

                                  "FROM dba WHERE FROM_TYPE='Do'" );

                           while(rsDo.next())
                {
                    if(rsDo.getRow() < NrowsDo)
                    outputDo += rsDo.getString("TO_TYPE")+ ", ";
                    else
                    outputDo += rsDo.getString("TO_TYPE");
                 }
                rsDo.close();
                statement.close();
                jTextArea4.append(outputDo+"};\n");
                }

//----------------------COMPENSATED-------------------------------------------------------
            ResultSet rsCo;
            String outputCo="";
            statement = connection.createStatement();
           ResultSet resultSetCo = statement.executeQuery("SELECT
           COUNT(*) FROM" +" (SELECT DISTINCT TO_TYPE FROM dba
           WHERE FROM_TYPE='Co')");
              // Get the number of rows from the result set
                 resultSetCo.next();
                 int NrowsCo = resultSetCo.getInt(1);
                    resultSetCo.close();
                    statement.close();

                    if (NrowsCo>=1)
                    {
                      jTextArea4.append("              (state=Co):{");
                    statement = connection.createStatement();
        rsCo = statement.executeQuery( "SELECT DISTINCT TO_TYPE " +
                            "FROM dba WHERE FROM_TYPE='Co'" );
                    while(rsCo.next())
                    {
                      if(rsCo.getRow() < NrowsCo)
                      outputCo += rsCo.getString("TO_TYPE")+ ", ";
                      else
                      outputCo += rsCo.getString("TO_TYPE");
                     }
                    rsCo.close();
                    statement.close();
                    jTextArea4.append(outputCo+"};\n");
                    }
//
    ----------------------SUSPENDED------------------------------------------------------
              ResultSet rsSu;
              String outputSu="";
              statement = connection.createStatement();
              ResultSet resultSetSu = statement.executeQuery("SELECT
              COUNT(*) FROM" +" (SELECT DISTINCT TO_TYPE FROM
              dba WHERE FROM_TYPE='Su')");
              // Get the number of rows from the result set
                            resultSetSu.next();
                            int NrowsSu = resultSetSu.getInt(1);
                                 resultSetSu.close();
                                 statement.close();
                                 if (NrowsSu>=1)
```

```
                                        {
                   jTextArea4.append("              (state=Su):{");
                                   statement = connection.createStatement();
                                   rsSu = statement.executeQuery( "SELECT
                                  DISTINCT TO_TYPE " +          "FROM dba
                                  WHERE FROM_TYPE='Su'" );
                                          while(rsSu.next())
                                        {
           if(rsSu.getRow() < NrowsSu)
           outputSu += rsSu.getString("TO_TYPE")+ ", ";
                   else
           outputSu += rsSu.getString("TO_TYPE");
                                        }
                 rsSu.close();
                 statement.close();
        jTextArea4.append(outputSu+"};\n");
                                        }
                                         }
                                        catch ( SQLException sqlex ) {
                                          sqlex.printStackTrace();
                                        }
                 jTextArea4.append("  (state=En):{Na};\n");
                    jTextArea4.append(" 1:state;\n"+"esac;\n\n");
                                    }
                                  } // end anonymous inner class
                                ); // end call to addActionListener
    }
    return buttonCONVERT;
}
```