UNIVERSITY of CALIFORNIA
Santa Barbara

Formal Specification and Verification of Asynchronously Communicating Web Services

A Dissertation submitted in partial satisfaction of the

requirements for the degree

Doctor of Philosophy

in

Computer Science

by

Xiang Fu

Committee in charge:

Professor Jianwen Su, co-Chair
Professor Tevfik Bultan, co-Chair
Professor Oscar H. Ibarra
Professor Ambuj K. Singh

September 2004

The dissertation of Xiang Fu is approved.

Professor Oscar H. Ibarra

Professor Ambuj K. Singh

Professor Jianwen Su, Committee co-Chair

Professor Tevfik Bultan, Committee co-Chair

June 2004

Formal Specification and Verification of Asynchronously Communicating Web Services

by

Xiang Fu

To my wife,

Zhengying Cao,

the only person worthy of my company.

Acknowledgements

First and foremost, I would like to express my gratitude to my advisors Professor Jianwen Su and Professor Tevfik Bultan, for their patience, support, and encouragement throughout my graduate studies. This dissertation would not have been possible without their invaluable advice and guidance. I am grateful to Professor Oscar Ibarra, for the collaboration work on counter machines, which becomes an integral part of this dissertation. I also want to thank Professor Ambuj Singh. His patience, time and feedback in this endeavor will be always appreciated. A special thank goes to Dr. Richard Hull at Bell Lab, who initially proposed the problem of automatic verification for Vortex workflows, which eventually evolved into my dissertation topic.

I have had the great pleasure to work with my fellow students and friends. I would like to thank Tuba Yavuz-Kahveci,Constantinos Bartzis, Aysu Betin-Can, Yujun Wang, Wei Niu, Jingyu Zhou, Tolga Can, Dan Koppel, Hoda Mokhtar, and Bin Lin, for their valuable suggestions and discussions during the process.

I wish to extend my deepest gratitude to my parents, for their years of hard work and dedicatation. Lastly, but most importantly, no words can express my gratitude to my wife, Zhengying Cao, for her unrelenting support, understanding and love.

Curriculum Vitæ

Xiang Fu

Personal

Born	November 16, 1977
	Shanghai, P.R. China

Education

1995–1999	B.S. in Computer Science
	Fudan University
	Shanghai, P.R. China

Publications:

Journal Articles:

(1) X. Fu, T. Bultan and J. Su. "Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services." *To appear* in a special issue of *Theoretical Computer Science* on CIAA'03.

Refereed Conference/Workshop Papers:

(2) X. Fu, T. Bultan and J. Su. "Realizability of Conversation Protocols with Message Contents", *To appear in Proceedings of the 2004 IEEE International Conference on Web Services (ICWS)* , 2004.

(3) X. Fu, T. Bultan and J. Su. "Model Checking XML Manipulating Software." *To appear in Proceedings of the 2004 International Symposium on Software Testing and Analysis (ISSTA)* , 2004.

(4) X. Fu, T. Bultan and J. Su. "WSAT: A Tool For Formal Analysis of Web Services." *Tool paper, to appear in Proceedings of the 16th International Conference on Computer Aided Verification (CAV)* , 2004.

(5) X. Fu, T. Bultan and J. Su. "Analysis of Interactive BPEL Web Services." *Proceedings of the 13th International World Wide Web Conference (WWW)*, pp. 621 – 630. New York, May 2004.

(6) X. Fu, T. Bultan and J. Su. "A Top-Down Approach to Modeling Global Behaviors of Web Services." *Workshop on Requirements Engineering and Open Systems (REOS)*, Monterey, CA, September 2003.

(7) X. Fu, T. Bultan and J. Su. "Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services." *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA)*, LNCS 2759, pp. 188 – 200, Springer. Santa Barbara, July 2003.

(8) T. Bultan, X. Fu, R. Hull and J. Su. "Conversation Specification: A New Approach to Design and Analysis of E-Service Composition." *Proceedings of the 12th International World Wide Web Conference (WWW)*, pp. 403 – 410. Budapest, Hungary, May 2003.

(9) X. Fu, T. Bultan and J. Su. "Formal Verification of E-Services and Workflows." *Proceedings of International Workshop on Web Services, E-Business, and the Semantic Web (WES), in conjunction with the 14th International Conference on Advanced Information Systems Engineering (CAiSE)*, LNCS 2512, pp. 188 – 202, Springer. Toronto, Canada, May 2002.

(10) X. Fu, T. Bultan, R. Hull and J. Su. "Verification of Vortex Workflows." *Proceedings of the 7th International Conference on Tools and Algorithms for*

the Construction and Analysis of Systems (TACAS), LNCS 2031, pp. 143 – 157, Springer. Genova, Italy, April 2001.

Abstract

Formal Specification and Verification of Asynchronously Communicating Web Services

by

Xiang Fu

As web services are paving the way to the next generation of electronic commerce, how to ensure design correctness for critical web services has been an important issue. This dissertation develops various automatic verification and analysis techniques for the validation of asynchronously communicating web services.

The formal verification of web services faces several special challenges: (1) lack of a formal model to characterize a composition of web services, (2) the undecidability of LTL verification due to the asynchronous communication, and (3) the expressive XPath based XML data manipulation, which is not supported directly by model checkers.

We establish a simple automata-theoretic model to study the global behaviors of a web service composition. Each individual web service (a peer) is specified using a finite state automaton, and is equipped with an unbounded FIFO queue to store incoming messages. The notion of conversations is developed to characterize global behaviors. Each conversation is a sequence of messages that are exchanged among peers, recorded in the order in which they are sent. Linear Temporal Logic (LTL) is naturally extended to specify desired properties on conversations.

We show that, due to the asynchronous communication, LTL verification for an arbitrary web service composition is undecidable.

To avoid the complexity caused by asynchrony, a synchronizability analysis is developed for web service compositions. Sufficient conditions are proposed to identify synchronizable web service compositions which generate the same set of conversations under both the synchronous and asynchronous communication semantics. Obviously, the LTL verification for synchronizable web service compositions can be conducted using the synchronous communication semantics. A similar realizability analysis is developed for a top-down specification approach based on conversation protocols.

A Guarded Automata (GA) model is developed to specify web services with XML data. Algorithms are developed to translate Guarded Automata to Promela (the input language of model checker SPIN), including the handling of XPath based data manipulation operations. This allows verification of web services at a great level of detail. Analyses and verification algorithms presented in this dissertation are implemented in the Web Service Analysis Tool (WSAT).

Contents

List of Figures

Chapter 1

Introduction

Web services [4, 32, 54, 67] are paving the way to the next generation of electronic commerce, because they are able to support automatic discovery and convenient integration of services, regardless of implementation platforms and across boundaries of business entities. For critical web services, where multi-million dollar transactions are carried out every day, any design error can cause potentially great losses, and ad-hoc repairs after failures are not acceptable. Hence it is desirable to statically ensure the correctness of web service designs before services are deployed. The goal of this research is to build an automatic verifier which can prove or disprove a web service design (or a composition of multiple web services) will satisfy a certain set of preset service properties.

Model checking [23] is one of the most promising techniques to achieve the above goal. Since behavior signatures (control flows and data manipulation semantics) of web services are published using standards such as BPEL4WS [12], it is possible to construct a formal model for each web service design and feed

it to a model checker. Desired properties, e.g., "eventually something good will happen" and "some bad event never happens", can be conveniently expressed using temporal logic [31]. Given a formal model and its desired properties, model checking conducts an *exhaustive* exploration (either explicitly or symbolically) of all possible behaviors of the model. Compared with other validation approaches such as testing, model checking gives designers absolute confidence when desired properties are verified. In addition, when a desired property is not satisfied, model checking will generate an error-trace which shows, step by step, how the property is violated by the model. Considering that web services are essentially distributed system and that designers of a composite web service sometimes do not have control over every component, it will be extremely hard to repeat the same error even if testing has reported a bug. The exact error trace reported by model checking is a big advantage over testing, because the error trace provides invaluable information to designers for understanding and removing bugs.

The formal verification of web services, however, is rather different than the verification of a general software system, due to the special characteristics of web services. To understand the special challenges that this work faces, in the following, we give a short introduction of the web service technology.

1.1 Web Services

While browser based web applications have been very successful in electronic commerce, for Business to Business (B2B) applications, the difficulty of integrating business processes across heterogeneous platforms has been a major hurdle in creating value-added composite services by integrating existing services.

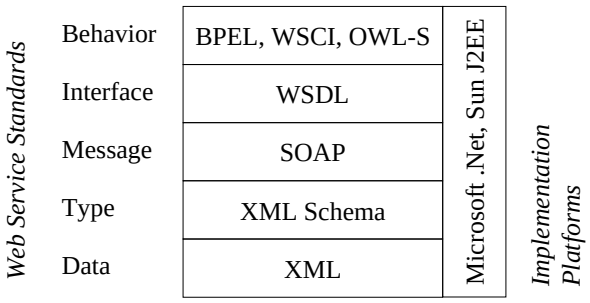


Figure 1.1. Web Service Standards Stack

The emerging web service technology is designed to overcome the challenge of automating business process interactions. The key is the use of a stack of standardized protocols (as shown in Fig. 1.1). For example, data transmission among web services is always wrapped using XML [26] format, so that services implemented on different platforms can communicate using a common language. XML Schema [82] provides the type system for XML documents, and SOAP [74] is a framework to further standardize the structured type declaration for XML messages. Web services themselves are described using public standards. Each web service has to publish its invocation interface, e.g., network address, ports, functions provided, and the expected XML message format to invoke the service, using the WSDL [80] standard. The behavior signature of each web service, i.e., control flows, data manipulation semantics and service qualities, can be described using one of many competing standards such as BPEL4WS [12], WSCI [79], and OWL-S [24]. The specification and functionality description of each web service are registered in a UDDI [75] registry, which allows each web service to be discovered by other services. Although web services can be implemented using different enterprise solutions, e.g., Microsoft .Net [60], J2EE [77], and embedding Java in executable BPEL4WS processes [25], the use of standardized protocols

allows automatic discovery, invocation and composition of web services regardless of implementation platforms and languages.

Web services are essentially distributed systems, however, departing from traditional tightly coupled systems like CORBA [68], web services are *loosely* coupled. Communication among web services is *asynchronous* – for each message exchanged, its sender and receiver do not have to synchronize the send and the receive actions. A message is stored in the message buffer of its receiver before it is consumed and processed. For each web service, its message buffer (usually a FIFO queue) is provided by the underlying message delivery platform such as Java Message Service (JMS) [52] and Microsoft Message Queuing Service (MSMQ) [64]. Such asynchronous messaging is very common for web services, especially for those (e.g. loan processing services) which require human intervention that may take minutes even days to complete. In addition, asynchronous communication, as put by Adam Bosworth, "is robust in the face of failure or delay. If another application happens to be unavailable or taking a long time, the system will still work. No application will become a single point of failure in the system. It is also robust in the face of change or diversity" [11].

1.2 Contributions

The characteristics of the web service technology have led to several challenges facing the effort of model checking web services: **(1)** Although numerous competing web service standards have been and are being proposed by the industry, many fundamental problems are not well defined and well understood. For example, one question is how to characterize the global behaviors of web service

compositions? Should both "send" and "receive" events be modeled? Another interesting question is how to extend temporal logics to reason about global behaviors of web service compositions? We need to resolve these questions in order to construct a formal model for web service compositions. **(2)** As we mentioned earlier, asynchronous communication is one of the advantages of the web service technology over traditional tightly coupled systems. This nice property, however, makes most interesting problems in analyzing web service compositions undecidable. Based on an earlier result [13], we can show that even if XML data semantics are abstracted away and each web service is modeled using a simple finite state machine, the composition of these asynchronously communicating finite state machines is Turing equivalent. It is not hard to see that the general problem of LTL verification for asynchronous composition of web services is undecidable [39]. **(3)** While the strength of web services relies on the use of XML, the tree-structured XML data and the expressive XPath based data manipulation are not supported directly by model checkers. Earlier efforts in this area, for example, the use of LTSA to model check BPEL4WS web services [36] and the petri-net approach to analyze web services [65], have concentrated on the abstract control flows of web services only, and data semantics were abstracted away.

This dissertation tackles the above challenges in the following ways.

(1) An Automata-Theoretic Modeling Approach: A simple and formal specification framework is developed to specify and reason about the global behaviors of a composition of web services. A web service composition is a closed system which consists of a finite set of communicating web services. Each individual web service (a peer) is specified using a Finite State Automaton (FSA),

and is equipped with an unbounded FIFO queue to store incoming messages. To characterize global behaviors of a web service composition, we assume there is a virtual watcher which listens silently to the network and records every send event. A conversation is a sequence of send events recorded by the watcher, where at the end of the run that generates this sequence, each peer stays in a final state and each input queue is empty. The notion of a conversation defines a "good global behavior" where each peer executes correctly according to its automaton specification, and there is no loss of information (i.e., each message ever sent is eventually consumed by its receiver). Clearly the set of "good behaviors" can be captured by the conversation set of a web service composition. Linear Temporal Logic (LTL) can be naturally extended to specify desired properties on conversations.

Based on the simple automata-theoretic model, we have several interesting theoretical observations. For example, the conversation set of an arbitrary web service composition is always context sensitive, and it is always closed under projection and join [16]. The expressiveness of bottom-up specified web service compositions motivates a top-down specification approach called conversation protocol. A conversation protocol [16, 39] is a single FSA which specifies the desired set of conversations but does not specify the implementation details of each peer. A conversation protocol, though weaker than a bottom-up specified web service composition, has several nice properties during analysis [16]. However, not every conversation protocol is realizable, i.e., some conversation protocol may not have a corresponding web service composition which generates exactly the same set of conversations as specified by the protocol. This dissertation gives a set of sufficient conditions to identify realizable conversation protocols.

To model real-world web services with XML data semantics, the automata-theoretic model is extended to a Guarded Automata (GA) model [41]. Each GA can have XML message contents and a finite set of XML local variables. Each transition of a GA is strengthened with an XPath guard, which determines both the transition condition as well as the assignments over message contents and local variables. GA is a very powerful model, and most static BPEL4WS web services can be translated into GA without any loss of data semantics [40].

(2) Synchronizability and Realizability Analyses: For bottom-up specified web service compositions, the asynchronous communication and unbounded input queues cause the undecidability of LTL verification, even if each peer is a standard FSA without data semantics. We develop a special analysis called "synchronizability analysis" [40] to avoid the undecidability. A set of sufficient synchronizability conditions are proposed to restrict control flows of each peer in a web service composition. When these conditions are satisfied, a web service composition is synchronizable, i.e., it generates the same set of conversations under both asynchronous and synchronous communication semantics. Since LTL properties are defined over conversations, and the synchronous composition of FSA peers is their Cartesian product (which is also an FSA that recognizes a regular language), the LTL verification for a synchronizable web service composition can be simply conducted using the synchronous communication semantics. In addition, if each peer is specified using a Deterministic Finite State Automaton (DFSA), and if the synchronous composition does not have deadlock, the asynchronous composition of all peers is guaranteed to be free of deadlock and unspecified message receptions. This guarantees a safe LTL verification, because all global behaviors generated will be "good" behaviors that are captured by

conversations.

There is a similar analysis called "realizability analysis" [39] for top-down specified conversation protocols. We have a set of sufficient realizability conditions to identify a realizable conversation protocol. The realizability analysis allows a 3-step specification and verification strategy: (1) a conversation protocol is specified using a realizable FSA, (2) desired LTL properties are verified on the conversation protocol, and (3) peer implementations are synthesized from the protocol.

In [42], we extend the realizability analysis to the Guarded Automata model, and synchronizability analysis is extended in a similar way. We have several interesting observations. For example, the realizability of a GA conversation protocol (i.e., the protocol is specified using a Guarded Automaton) does not depend on the realizability of its skeleton (which is generated by dropping message contents and transition guards from the GA). However, a GA conversation protocol is guaranteed to be realizable if its skeleton satisfies the realizability conditions and one additional condition. We also develop iterative refined analysis for one of the realizability conditions called "autonomous condition", and symbolic analysis algorithms are designed for other conditions.

(3) Handling of XML Data Manipulation: To completely verify web services with XML data semantics, web service designs specified using popular industry standards (e.g. BPEL4WS) are first translated into an intermediate representation called Guarded Automata (GA) [40]. Then GA are translated into Promela processes [41], where Promela is the input language of model checker SPIN [50]. SPIN is used as our back-end model checker to conduct the LTL

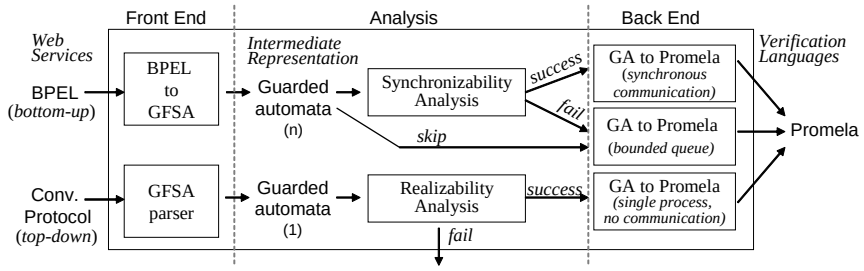


Figure 1.2. WSAT architecture

verification. The translation from BPEL4WS to GA is straightforward, however, the translation from GA to Promela has to deal with many tricky issues. We have to first develop a type mapping from XML Schema to the type system of Promela. Then based on this type mapping, algorithms are developed to translate an XPath expression to a segment of Promela code. For each XPath expression, its Promela translation is essentially a symbolic execution of that XPath expression over an XML Schema type. Special handling has to be paid to XML Schema types with multiple occurrences, function calls like `position()` and `last()`. For example, each appearance of `position()` function will be mapped into a corresponding integer variable. During the symbolic execution of the XPath expression, the value of that integer variable will be carefully updated so that when the `position()` is called the corresponding integer variable contains the right return value. The handling for `last()` is even more complicated.

The project Web Service Analysis Tool (WSAT) implements and integrates all[1] the analysis and verification algorithms presented in this dissertation. Fig. 1.2 presents the general architecture of WSAT. The front-end of WSAT accepts input specified using popular industry standards such as BPEL4WS and WSDL, as well as top-down specified conversation protocols. Then BPEL4WS web ser-

[1]At the moment of filing this dissertation, implementation has not included the symbolic analysis algorithms in Chapter 4.

vices are translated into GA, where synchronizability analysis (or realizability analysis) is applied. GA are then translated into Promela, where SPIN is called to conduct LTL verification at the back-end. The use of Guarded Automata as the intermediate representation allows a very flexible architecture of WSAT. More web service specification languages (e.g. WSCI and OWL-S) can be supported at the front-end, and more model checking modules (e.g. symbolic verification modules) can be added at the back-end, without changing the front-end.

1.3 Organization

The rest of the dissertation is organized as follows. Chapter 2 introduces a simple and abstract model of web service composition, and several interesting theoretical observations are provided. Chapter 3 presents the synchronizability and realizability analyses, which help to avoid the undecidability caused by asynchrony during verification. Chapter 4 extends the model to a GA model, and discusses the realizability analysis in the extended model. Chapter 5 discusses the expressive power hierarchy of web service compositions, which is influenced by the arithmetic constraints used in guards and the interconnection pattern of peers. Chapter 6 presents the GA model with XML data semantics, and discusses the translation from BPEL4WS standard to the GA model. Chapter 7 presents how to handle XML data manipulation in LTL verification, and briefly introduces the Web Service Analysis Tool (WSAT). Chapter 8 concludes.

Chapter 2

A Formal Model for Web Service Compositions

This chapter presents a simple specification framework [16, 39, 40] to formally model the composition of interacting web services. A *web service composition* is a *closed*[1] system where a finite set of interacting (individual) web services communicate with each other via asynchronous messaging. We consider the problem of how to characterize the global behaviors generated by a web service composition, as well as how to reason about their correctness, e.g., can they meet a certain preset system goal that is expressed using temporal logic? We provide several initial theoretical observations of the model, which motivate the realizability and synchronizability analyses discussed in later chapters. Note that the model presented in this chapter is a *contentless* one, where XML message contents are

[1] In our earlier work [16, 39, 40], a web service composition is denoted using terms "composite e-service" or "composite web service". In this dissertation, we distinguish the concept of open system from closed system, depending on whether the system exchanges messages with outside world. A closed system is called a "composition", and an open system is called a "composite service".

abstracted away. The model will be extended in Chapter 4 and 6 to address this problem.

In our framework, an observable global behavior of a web service composition is captured by the notion of a conversation, and the set of all observable behaviors forms the conversation set. A *conversation* [16] is the global sequence of messages that are exchanged among peers, recorded in the order in which they are sent. Such message-oriented behavior modeling is not only simple, but more importantly, it requires web services to reveal the least amount of information that is necessary to make meaningful compositions. Thus complex internal states (e.g. in legacy systems) can be hidden. In addition, conversations immediately permit the usage of Linear Temporal Logic (LTL) [31] to express properties of web service compositions [39].

We have some interesting observations of the conversation set of a web service composition. For example, it is shown in [16] that the conversation set of a web service composition (where each peer is specified using a standard finite state automaton) is always context sensitive but may not be context-free. Later, in [39], LTL model checking for a web service composition is proved to be undecidable. In [16], we also present a closure property for conversation sets, which plays an important role in the comparison of the expressive power of the conversation oriented framework and Message Sequence Chart (MSC) graphs [7].

The high complexity associated with analyzing bottom-up specified web service compositions prompts an alternative top-down specification approach. In [16, 39], we propose the notion of a conversation protocol, which specifies the set of desired global behaviors of a web service composition, given its intercon-

nection pattern. While most industry web service standards (e.g. WSDL [80], BPEL4WS [12], and WSCI [79]) favor the bottom-up fashion, a conversation protocol does resemble, however still differs from, industry initiations such as IBM Conversation Support [51] and MSC [63]. The difference concerning the modeling perspective between a conversation protocol and an MSC leads to rather different decidability results in the realizability analysis for the two modeling approaches.

The chapter is organized as follows. We will first introduce the general notion of a composition schema, which specifies the static interconnection pattern of a web service composition. Then we discuss the specification of each peer, i.e., each participant of a web service composition. Next we discuss how to characterize global behaviors of a web service composition, and introduce the notion of a conversation. We then present several theoretical observations on conversation sets, motivated by which, we propose the top-down specification approach called conversation protocol. We give a short discussion of related work, and present a comparative study of MSC. Finally we study a variation of our model for reactive web services which have behaviors of infinite length.

2.1 A General Composition Architecture

In this section we describe a paradigm for modeling web service compositions. While abstract and focusing primarily on global behaviors, our paradigm is based on the fundamental constructs of the web services as promoted by, e.g., BPEL4WS [12], WSCI [79], IBM's Web services Toolkit [76], Microsoft's .Net [60], Java Message Service [52], Microsoft Message Queuing Service [64] and other industrial products and proposals. It also follows the model adopted by much of the

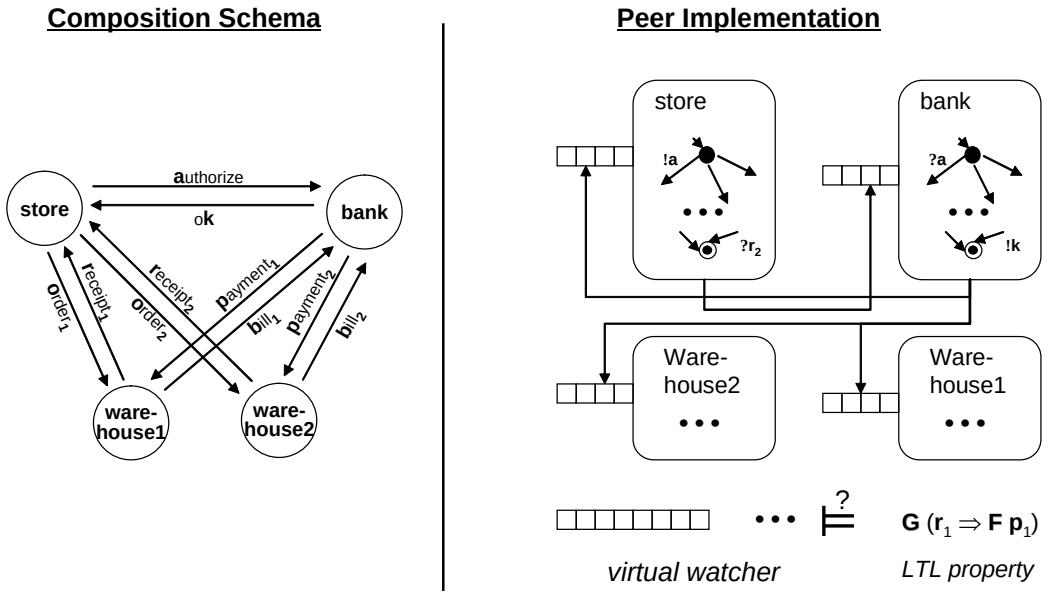


Figure 2.1. A Warehouse Web Service Composition

research on web service composition [28, 65], work on web service programming languages [5, 35], and the AZTEC prototype [22].

Fig. 2.1 shows a natural (though informal) way to specify a composition of web services. For each web service composition, its specification consists of two parts: a composition schema and a set of peer implementations. A composition schema specifies the set of peer prototypes and the set of message classes exchanged among peers. Each peer implementation describes the (abstract) control flows of each individual peer prototype. As communication is asynchronous, each peer is equipped with a FIFO queue to store incoming messages. We assume that there is a *virtual watcher* which records the sequence of messages as they are *sent* by the peers. The sequence of messages recorded by the watcher is called a conversation. (Note that the virtual watcher is a construct we use to reason about the interactions among different peers and it is not implemented.) A conversation

can be regarded as a linearization of the message events, similar to the approach used in defining the semantics of Message Sequence Charts [63] in [7]. We now formalize the above descriptions with the definitions below.

Definition 2.1 *A* composition schema *is a tuple* (P, M) *where* $P = \{p_1, \ldots, p_n\}$ *is a set of* peer prototypes*, and the* alphabet M *is a set of message classes. Each peer prototype* $p_i = (M_i^{in}, M_i^{out})$ *is a pair of disjoint sets of message classes* $(M_i^{in} \cap M_i^{out} = \emptyset)$*, and let* $M_i = M_i^{in} \cup M_i^{out}$ *be the alphabet of* p_i*. M satisfies the following:*

$$\bigcup_{i \in [1..n]} M_i^{in} = \bigcup_{j \in [1..n]} M_j^{out} = M$$

Implied by the above definition, each message class is transmitted on a single directional peer to peer channel, and a peer cannot send a message back to itself. At this moment, we assume that message classes do not have contents. But later in Chapter 6, we will show that each message class can be associated with a type for its contents which is declared using XML Schema [82].

Definition 2.2 *A* web service composition *is a tuple* $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$*, where* (P, M) *is a conversation schema,* $n = |P|$*, and each* $\mathcal{A}_i$ *is the* peer implementation *(or simply "peer") for the corresponding peer prototype* $p_i = (M_i^{in}, M_i^{out}) \in P$*. The alphabet of* $\mathcal{A}_i$ *is* $M_i = M_i^{in} \cup M_i^{out}$*.*

The above definition is a general notion of web service compositions where we do not specifically restrict the expressive power of peer implementations. In this chapter, we will study two types of web service compositions whose peers are specified using standard Finite State Automata (FSA) and Büchi Automata,

respectively. In later chapters, we will also introduce web service compositions whose peers are specified using Guarded Automata (to bring in the message contents).

A web service composition is also called a "composition" for short. For different types of web service compositions, the naming rule is based on the automata that are used to specify peers. For example, the three types of web service compositions mentioned above will be called "FSA composition", "Büchi composition", and "GA composition", respectively.

Example 2.1 Fig. 2.1 shows a web service composition that consists of four peers: a retail store that plans to replenish its inventory, its bank, and two warehouses that supply goods. In a (simplified) typical scenario (where the peer implementation will be given in the next section), the store requests an authorization from the bank; after receiving the approval from the bank, the store can send one or more orders to the warehouses. When a warehouse receives an order, it responds by billing the bank for the amount on the order, and sends the store a receipt. The bank, in turn, makes a payment after receiving a bill. Clearly, let $S = (P, M)$ be the composition schema of Fig. 2.1, $|P| = 4$, and $|M| = 10$. For peer prototype store, its input alphabet is $\{\mathsf{ok}, \mathsf{receipt}_1, \mathsf{receipt}_2\}$, and its output alphabet is $\{\mathbf{a}\mathsf{uthorize}, \mathsf{order}_1, \mathsf{order}_2\}$. In the rest of the chapter, to save space, we represent each message class using the bold letter in its name. For example, **k** stands for o**k**, and **a** stands for **a**uthorize. ∎

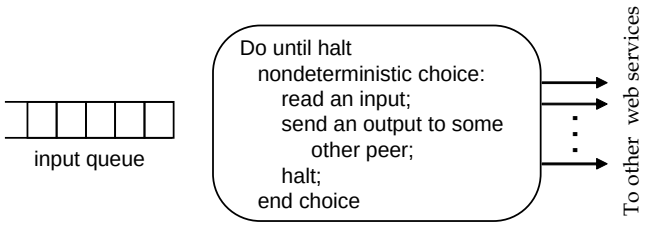


Figure 2.2. The Model of A Peer

2.2 Peers

Fig. 2.2 illustrates an abstraction of an individual web service (called here a *peer*). Roughly, a peer can be viewed as a "program" that decides, based on the received messages and the messages already sent, if a new message should be sent, and/or if the session should terminate. Note that each peer is equipped with a FIFO queue to store incoming messages, as the communication among web services is asynchronous. This modeling resembles industry messaging platforms such as JMS [52] and MSMQ [64], which provide XML message delivery.

In the following technical discussions, we consider a special family of peers called "FSA peers"[2], where each peer is described using a standard nondeterministic Finite State Automaton (FSA). The simplicity of FSA allows very interesting theoretical observations. On the other hand, as we will show in Section 6, when extended to the Guarded Automata model, our framework can capture most industry web service designs. At this moment, as message classes do not have contents, the FSA peer can be regarded as the abstract control flow of an individual web service design. This is illustrated by Fig. 2.3, where the message classes effectively dictate the actions to be taken by each peer and consequently

[2]An FSA peer is essentially the Mealy peer in [16].

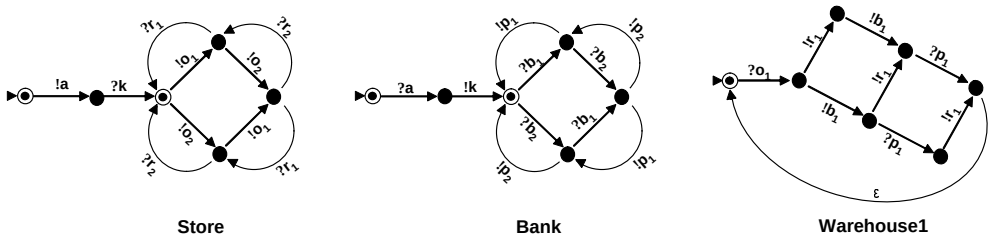


Figure 2.3. Peer Implementations for Fig. 2.1

the responses. Formally, an FSA peer is defined as follows.

Definition 2.3 *Let $S = (P, M)$ be a composition schema, $p_i \in P$ be a peer prototype, and let $p_i = (M_i^{in}, M_i^{out})$. An* FSA peer $\mathcal{A}_i$ *which* implements p_i *is a standard nondeterministic Finite State Automaton $\mathcal{A}_i = (M_i, T_i, s_i, F_i, \delta_i)$ where $M_i = M_i^{in} \cup M_i^{out}$. Here T_i is a finite set of states, $s_i \in T$ is the initial state, $F_i \subseteq T$ is a set of final states, and $\delta_i : T_i \times (M_i \cup \{\epsilon\}) \rightarrow 2^{T_i}$ is a transition relation. A transition $\tau \in \delta_i$ can be one of the following three types: (1) a send-transition of the form $(t_1, !\alpha, t_2)$ which sends out a message $\alpha \in M_i^{out}$, (2) a receive-transition of the form $(t_1, ?\beta, t_2)$ which consumes a message $\beta \in M_i^{in}$ from its input queue, and (3) an ϵ-transition of the form (t_1, ϵ, t_2).*

Definition 2.4 *A web service composition is an* FSA web service composition *if all of its peers are FSA peers.*

Example 2.2 Fig. 2.3 shows the peer implementation for each peer prototype in Fig. 2.1. (The implementation for Warehouse2 is analogous to the implementation for Warehouse1.) Studying the peer implementation of Warehouse1, it is not hard to see that the timing of sending **r**eceipt$_1$ from Warehouse1 to Store is independent of the timing of the corresponding messages **b**ill$_1$ and **p**ayment$_1$ between Bank

and Warehouse1. Obviously, we can have a more restrictive implementation for Warehouse1 by enforcing the sending of $\mathbf{p}\text{ayment}_1$ after $\mathbf{b}\text{ill}_1$. ∎

Remark: Given an FSA peer, if we drop all "!" and "?" from its transitions, and the resulting FSA (which can be regarded as a standard FSA that accepts words, instead of generating words) is a Deterministic Finite State Automaton (DFSA), then we call such a peer a *DFSA peer*. For example, each peer in Fig. 2.3 is a DFSA peer. ∎

2.3 Conversations

We now formally define the notion of conversations to capture the global observable behaviors of an FSA composition. Let $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n) \rangle$ be an FSA composition where $n = |P|$, its *global configuration* (or simply configuration) is a $(2n+1)$-tuple of the form

$$(Q_1, t_1, ..., Q_n, t_n, w)$$

where for each $j \in [1..n]$, $Q_j \in (M_j^{in})^*$, $t_j \in T_j$ and $w \in M^*$. Here t_i, Q_i are the local state and queue contents of peer $\mathcal{A}_i$ respectively, and w is the global watcher content at this configuration. For each configuration c_i, we use $gw(c_i)$ to denote its watcher content.

Let the set of states, transition relation, and etc. of a peer $\mathcal{A}_i$ be all labeled with subscript i, e.g., δ_i is the transition relation of peer $\mathcal{A}_i$. For two configurations $c = (Q_1, t_1, ..., Q_n, t_n, w)$ and $c' = (Q'_1, t'_1, ..., Q'_n, t'_n, w')$, we say that c *derives* c', written as $c \rightarrow c'$, if one of the three types of *actions* is executable:

- (Peer $\mathcal{A}_j$ executes an ϵ-move) there exists $1 \leq j \leq n$ such that

  1. $(t_j, \epsilon, t'_j) \in \delta_j$,
  2. $Q'_j = Q_j$,
  3. for each $k \neq j$, $Q'_k = Q_k$ and $t'_k = t_k$, and
  4. $w' = w$.

- (Peer $\mathcal{A}_j$ consumes an input) there exists $1 \leq j \leq n$ and $\alpha \in M_j^{in}$ such that

  1. $(t_j, ?\alpha, t'_j) \in \delta_j$,
  2. $Q_j = \alpha Q'_j$,
  3. $Q_k = Q'_k$ for each $k \neq j$,
  4. $t'_k = t_k$ for each $k \neq j$, and
  5. $w' = w$.

- (Peer $\mathcal{A}_j$ sends an output to peer $\mathcal{A}_k$ and writes to the watcher) there exists $1 \leq j, k \leq n$ and $\beta \in M_j^{out} \cap M_k^{in}$ such that

  1. $(t_j, !\beta, t'_j) \in \delta_j$,
  2. $Q'_k = Q_k \beta$,
  3. $Q'_l = Q_l$ for each $l \neq k$,
  4. $t'_l = t_l$ for each $l \neq j$, and
  5. $w' = w\beta$.

The above three actions can be denoted by $c \xrightarrow{\epsilon} c'$, $c \xrightarrow{?\alpha} c'$, and $c \xrightarrow{!\beta} c'$, respectively. We denote by $\xrightarrow{*}$ the reflexive and transitive closure of $\rightarrow$.

Definition 2.5 *Let $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be an FSA web service composition, a configuration list $\gamma = c_0, c_1, \ldots, c_k$ is a* run *of $\mathcal{S}$ if it satisfies the first two of the following three conditions, and γ is a* complete run *if it satisfies all the three conditions.*

1. *$c_0 = (\epsilon, s_1, \ldots, \epsilon, s_n, \epsilon)$ is the initial configuration where s_i is the initial state of $\mathcal{A}_i$ for each $i \in [1..n]$.*
2. *for each $j \in [0..k-1]$, $c_j \rightarrow c_{j+1}$.*
3. *$c_k = (\epsilon, t_1, \ldots, \epsilon, t_n, w)$ is a final configuration where t_i is a final state of $\mathcal{A}_i$ for each $i \in [1..n]$.*

A word w over M is a conversation *of $\mathcal{S}$ if there exists a complete run γ' (let its last configuration be c') such that $w = gw(c')$. The* conversation set *of $\mathcal{S}$, written as $\mathcal{C}(\mathcal{S})$, is the set of all conversations for $\mathcal{S}$.*

Example 2.3 Let $\mathcal{S}_1$ be the FSA web service composition presented in Fig. 2.1 and Fig. 2.3. It can be verified that $\mathcal{C}(\mathcal{S}_1)$ can be captured by the following expression:

$$\mathbf{ak}\,\text{SH}((\mathbf{o}_1\,\text{SH}(\mathbf{r}_1, \mathbf{b}_1\mathbf{p}_1))^*, (\mathbf{o}_2\,\text{SH}(\mathbf{r}_2, \mathbf{b}_2\mathbf{p}_2))^*).$$

Here SH is an shuffle operator which, given two words w_1 and w_2, generates a set of new words by inserting each letter of w_1 into w_2 while maintaining their relative order in w_1. For example, $\text{SH}(\mathbf{r}_1, \mathbf{b}_1\mathbf{p}_1) = \{\mathbf{r}_1\mathbf{b}_1\mathbf{p}_1, \mathbf{b}_1\mathbf{r}_1\mathbf{p}_1, \mathbf{b}_1\mathbf{p}_1\mathbf{r}_1\}$. Obviously, SH can be extended to work on two word sets by taking the union of the results of applying the shuffle operator on any pair of words from the two word sets. ∎

We call a finite run *completable* if it is the prefix of a complete run. During a complete run, a peer is said to *terminate* at some configuration c_i if after c_i the peer does not take any transitions, and its local state at c_i is a final state. During a run, a peer is said to be *receptive* to a message α at configuration c_i if there is a transition which starts from the state of the peer at c_i and consumes α. During a run, a peer is said to be *stuck* at configuration c_i if the peer is not receptive to the message at the head of the input queue, and none of the transitions starting from the state of the peer at c_i is a send-transition or ϵ-transition. A configuration c_i is said to be a *deadlock* configuration, if there is no c_j such that $c_i \rightarrow c_j$, and at configuration c_i there is at least one peer not in final state.

Notice that the notion of "complete run" and "conversation" captures the "good behaviors" generated by a web service composition where each peer lawfully executes according to the FSA specification, and each message transmitted is eventually consumed. There might be "bad runs" where some peer gets stuck by an unexpected message, or the whole system gets into a deadlock. Hence for a web service composition $\mathcal{S}$, its conversation set $\mathcal{C}(\mathcal{S})$ does not really cover all possible behaviors of $\mathcal{S}$. But later as we show in Chapter 3, a synchronizability (realizability) analysis can guarantee that $\mathcal{C}(\mathcal{S})$ covers all possible behaviors.

2.4 Linear Temporal Logic

Now given the notion of conversations, it is easy to extend Propositional Linear Temporal Logic (LTL) [31] into the framework to describe desired properties of a web service composition. For a conversation $w = w_0, w_1, w_2, \ldots, w_n$, let w_i denote the i-th message in w, and $w^i = w_i, w_{i+1}, w_{i+2}, \ldots$ the i-th suffix of

w. Extending the definitions in [31] and [23], we define the LTL properties on conversations as follows. Given a composition schema (P, M), the set of atomic propositions is defined as the power set of message classes, i.e., $AP = 2^M$. LTL properties are constructed from atomic propositions in AP, logical operators $\wedge, \vee, \neg$, and LTL operators $\mathbf{X}, \mathbf{G}, \mathbf{U}, \mathbf{F}$. Given LTL formulas ϕ, and φ, an atomic proposition $\psi \in AP$, and a word $w \in M^*$ (let $w = w_0, \ldots, w_n$), the syntax and semantics of LTL formulas can be defined as follows:

$$
\begin{array}{lll}
w \models \psi & \text{iff} & w_0 \in \psi \\
w \models \neg\phi & \text{iff} & w \not\models \phi \\
w \models \phi \wedge \varphi & \text{iff} & w \models \phi \text{ and } w \models \varphi \\
w \models \phi \vee \varphi & \text{iff} & w \models \phi \text{ or } w \models \varphi \\
w \models \mathbf{X}\phi & \text{iff} & |w| > 1 \text{ and } w^1 \models \phi \\
w \models \mathbf{G}\phi & \text{iff} & \text{for all } 0 \leq i \leq n, w^i \models \phi \\
w \models \phi \mathbf{U} \varphi & \text{iff} & \text{there exists } 0 \leq j \leq n, \text{ such that } w^j \models \varphi \text{ and,} \\
 & & \text{for all } 0 \leq i < j, w^i \models \phi \\
w \models \mathbf{F}\phi & \text{iff} & w \models M\mathbf{U}\phi
\end{array}
$$

Intuitively, temporal operators **X**, **G**, **U** and **F** mean "next", "globally", "until", and "eventually", respectively. We say that a web service composition $\mathcal{S}$ satisfies an LTL formula ϕ, i.e., $\mathcal{S} \models \phi$, if and only if, for each conversation $w \in \mathcal{C}(\mathcal{S})$, $w \models \phi$.

2.5 Conversation Set is Context Sensitive

One natural question concerning a conversation set is: since each peer is specified using a finite state machine, is every conversation set a regular language?

The following is a counter example.

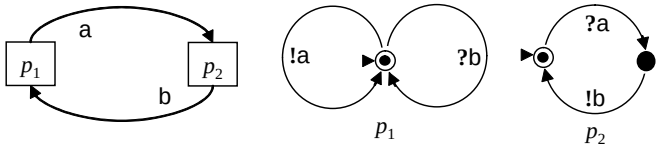


Figure 2.4. The FSA Composition for Example 2.4

Example 2.4 Fig. 2.4 shows an FSA composition $\mathcal{S}$ with two peers. Peer p_1 sends requests a while p_2 responds with one b message for each a message. Since a messages can be temporarily stored in the queue of p_2, the $\mathcal{C}(\mathcal{S})$ consists of words with the same number of a's as b's and each b has a corresponding a that occurs somewhere beforehand. Note that $\mathcal{C}(\mathcal{S}) \cap (a^*b^*) = \{a^n b^n \mid n \geq 0\}$, therefore $\mathcal{C}(\mathcal{S})$ is not regular. However, $\mathcal{C}(\mathcal{S})$ is context free because it can be recognized by a pushdown automaton. ∎

Interestingly, if an FSA composition is restricted to the synchronous communication mode (i.e., for each message, the corresponding send and receive action are synchronized, and there are no queues), it is easy to observe that its conversation set is always regular. However the assumption of synchronous communication or bounded buffer size may not work for web services. Web services are designed to communicate via asynchronous messaging, and messaging platforms like JMS and MSMQ can provide unbounded buffer size as long as there are enough system resources available.

Using an idea similar to that in Example 2.4, one can easily construct a three-peer FSA composition $\mathcal{S}'$ which generates a non-context-free conversation set as follows. Peer p_1' is a single state FSA which, from its initial state, can either

send out a message a to peer p_2' or receive a message c from peer p_3' (just like the p_1 in Fig. 2.4); peer p_2' sends a corresponding message b to peer p_3' for each a received; peer p_3' sends a message c to peer p_1' for each b received. Clearly $\mathcal{C}(\mathcal{S}') \cap (a^*b^*c^*) = \{a^n b^n c^n \mid n \geq 0\}$, hence $\mathcal{C}(\mathcal{S}')$ is context sensitive but not context free. The following theorem summarizes the above discussion.

Theorem 2.5 Let $\mathcal{S}$ be an arbitrary FSA composition.

(a) $\mathcal{C}(\mathcal{S})$ is always context sensitive.

(b) If the computations of conversations are restricted to only allow queues with length bounded by a fixed constant, then the restricted conversation set $\mathcal{C}^{\text{bounded}}(\mathcal{S})$ is regular.

Proof: To prove part (a), we can construct a multi-tape Linear Bounded Automaton (LBA) to simulate the FSA composition $\mathcal{S}$. The LBA has one tape for each input queue, and it simulates the transition of each peer. It is not hard to see that, given a word w, the LBA can accept w with the sum of the sizes of all its tapes bounded by the length of w. Since each LBA recognizes a context-sensitive language, it immediately leads to the conclusion in part (a). For part (b), we can simply construct an FSA to simulate the composition of all peers by encoding the queue contents into states (since queue length is bounded, the number of states is finite). It follows that $\mathcal{C}^{\text{bounded}}(\mathcal{S})$ is regular. ∎

Theorem 2.5 highlights differences between the synchronous communication models in I/O and interface automata and the asynchronous model described here. In fact, given a set of finite state peers, LTL model checking is undecidable.

Theorem 2.6 Given an FSA composition $\mathcal{S}$ and an LTL property ϕ, determining if $\mathcal{S} \models \phi$ is undecidable.

Proof: As a two-peer FSA composition is essentially a system of two Communicating Finite State Machines (CFSM) in [13], directly from the CFSM's Turing equivalence result in [13], for each Turing Machine TM we can construct a two-peer FSA composition $\mathcal{S}$ that simulates TM and exchanges a special message (say m_t) once TM terminates. Thus TM terminates if and only if the LTL formula $S \models M$ **U** $\{m_t\}$ is true. Here **U** is the temporal operator meaning "until" [23], and the meaning of the formula is "eventually message m_t will appear". ∎

2.6 Closure Properties of Conversation Sets

In this section, we introduce several interesting observations on the closure properties of a conversation set. We show that a conversation set is always closed under a special swapping operator named "prepone" , and it is closed under projection/join. However, the combination of these two operators is still not strong enough to capture every conversation set. We then introduce a "local prepone" operator, which, combined with join, can formulate a succinct mathematical characterization for the conversation set produced by any FSA web service composition. Notice that, the simple mathematical characterization, however, does not imply that to enumerate or to analyze the conversation set for an arbitrary FSA composition (e.g. to verify a safety property) is decidable.

2.6.1 Closure Under Prepone

We now return to the phenomenon exposed by Example 2.4. A close examination indicates that the primary reason for this behavior is that the message queue of a peer serves as a "buffer" for the input: while conversations monitor the arrival of messages at the queues, the messages may not be read right away. To understand this effect, we introduce a swapping operator PREPONE as follows.

Definition 2.6 *Let M be the alphabet of an FSA web service composition $\mathcal{S}$, operator* PREPONE *is a function from M^* to 2^{M^*}. Let $w = w'm_1m_2w''$ be a word in M^*, where m_1 is a message from p_i to p_j and m_2 is a message from p_x to p_y. If either (1) $\{p_i, p_j\}$ and $\{p_x, p_y\}$ are disjoint, or (2) $p_i = p_y$ and $p_j \neq p_x$, then* PREPONE(w) *includes the word $w'm_2m_1w''$.*

Intuitively, the operator PREPONE allows two messages in a conversation to be swapped if the senders and receivers are completely disjoint, or a later message to a peer can arrive in the queue earlier than an outgoing message from the peer, because the outgoing message cannot depend on a later arrived message. It is important to note that PREPONE applies to the global sequence of messages observed by the watcher. If L is a language over M, we define PREPONE(L) to be the smallest language that contains L and is closed under PREPONE. The following interesting property holds for PREPONE.

Lemma 2.7 For each web service composition $\mathcal{S}$, PREPONE$(\mathcal{C}(\mathcal{S})) \subseteq \mathcal{C}(\mathcal{S})$ (closure under PREPONE).

Proof: It suffices to show that given a conversation w, if $w' \in$ PREPONE(w), then w' is also a conversation. To prove the above argument, we can construct a

complete run γ' for w', by modifying the complete run γ for w.

Now suppose that $w = u\alpha\beta v$ and $w' = u\beta\alpha v$, where α and β are the pair of messages which satisfy the conditions to apply the prepone operator. Now let the complete run γ be written as

$$\gamma = c_0 \rightarrow \cdots \rightarrow c_i \xrightarrow{!\alpha} c_{i+1} \rightarrow \ \ldots \ \rightarrow c_k \xrightarrow{!\beta} c_{k+1} \rightarrow \ \ldots \ \rightarrow c_n.$$

Since the α and β are consecutive messages in the conversation, from configuration c_{i+1} to c_k, each action is either a consume or an ϵ transition. Note that consume and ϵ actions are internal transitions which do not affect other peers. We can permute the actions between c_{i+1} and c_k (however, keep the relative order of actions of the same peer), without affecting the rest of γ. The action sequence between c_{i+1} and c_k of γ will be re-arranged so that it consists of three subsequences: **seq1** where all actions are taken by the sender or the receiver of α; **seq2** where all actions are taken by a peer which is neither the sender/receiver of α, nor the sender of β; **seq3** where all actions are taken by the sender of β. Note that after action sequence is re-arranged, the contents of c_{i+1} to c_k should be adjusted accordingly.

Now by the two alternative conditions to apply the prepone operator, the sets of actions involved in **seq1**, **seq2**, **seq3** are pairwise disjoint. (Notice that in the second condition of Definition 2.6, i.e., "$p_i = p_y$ and $p_j \neq p_x$", the "$p_i = p_y$" makes **seq2** to be disjoint with **seq3**.)

Now we can construct γ' from γ. First, we copy the parts: c_0 to c_i, and c_{k+1} to c_n, from γ to γ'. Then the action sequence of the middle part (i.e., c_i to c_{k+1}) of γ' is the following:

$$\textbf{seq2 seq3} \ !\beta \ !\alpha \ \textbf{seq1} \tag{2.1}$$

Note that since **seq1**, **seq2**, and **seq3** are pairwise disjoint, the above swap of these subsequences is feasible. The rest of the construction is to properly reset the contents of configurations from c_i to c_{k+1} according to Equation 2.1. ∎

2.6.2 Closure Under Join

The second closure property of a conversation set concerns combining "local views" of peers into global conversations, this is reminiscent of the join operator in the relational database model.

Example 2.8 Consider a composition schema S that has four peer prototypes p_1, p_2, p_3, p_4, and three messages $p_1 \rightarrow p_2 : a$, $p_3 \rightarrow p_4 : b$, and $p_4 \rightarrow p_3 : c$. Is there any FSA composition on schema S that generates the regular language $\{a, bc\}$? Note that the peer groups $\{p_1, p_2\}$ and $\{p_3, p_4\}$ are in fact independent; there is no communication possible between them. Hence any FSA composition that generates $\{a, bc\}$ also generates each of ϵ, abc, bac, and bca. ∎

The above example suggests that if two global behaviors have exactly the same local views, they are indistinguishable. Next we formalize the concept of "projection" and "join" as below. For a composition schema $S = (P, M)$, given a word $w \in M^*$, $\pi_i(w)$ denote the projection of w to the alphabet M_i of the peer prototype p_i, i.e., $\pi_i(w)$ is a subsequence of w obtained from w by removing all the messages which are not in M_i. When the projection operation is applied to a set of words the result is the set of words generated by application of the projection operator to each word in the set. Given a composition schema (P, M) where $n = |P|$, let $L_1 \subseteq M_1^*, \ldots, L_n \subseteq M_n^*$, the join operator is defined as follows:

$$\text{JOIN}(L_1, \ldots, L_n) = \{w \mid w \in M^*, \forall i \in [1, n] : \pi_i(w) \in L_i\}.$$

It is not hard to infer the following:

$$L = \textsc{join}(L_1, \ldots, L_n) \;\Rightarrow\; \forall i \in [1..n] : \pi_i(L) \subseteq L_i. \tag{2.2}$$

Given a language $L \subseteq M^*$, the join closure is defined as follows:

$$\textsc{joinc}(L) = \textsc{join}(\pi_1(L), \ldots, \pi_n(L))$$

We have the following result:

Lemma 2.9 For each FSA composition $\mathcal{S}$: $\textsc{joinc}(\mathcal{C}(\mathcal{S})) \subseteq \mathcal{C}(\mathcal{S})$.

To prove the above lemma, we can show that for each word w in $\textsc{joinc}(\mathcal{C}(\mathcal{S}))$, and for each peer prototype p_i, there is a "complete local execution" of the peer for $\pi_i(w)$. Then we can construct a global run which simulates each local run, and generates w. The complete proof is essentially a part of the proof of Lemma 2.14. In fact, Lemma 2.9 is an implication of Lemma 2.14.

2.6.3 Combining Prepone and Join

We can combine the closure of prepone and join. Lemmas 2.7 and 2.9 immediately imply the following.

Lemma 2.10 Let $\mathcal{S}$ be an FSA composition and its alphabet is M. Given a language $L \subseteq M^*$, let *closure*(L) denote the minimal superset of L that is closed under PREPONE and JOINC. The following holds:

$$L \subseteq \mathcal{C}(\mathcal{S}) \;\Rightarrow\; \mathit{closure}(L) \subseteq \mathcal{C}(\mathcal{S})$$

Essentially, the above lemma states that if we know that a set of conversations can be generated by an FSA composition, then the closure of that set should

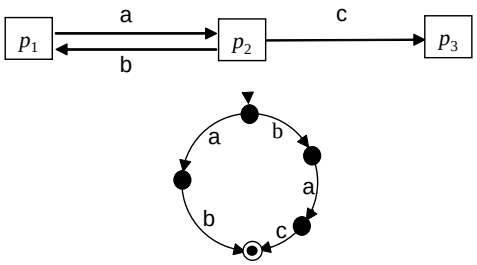


Figure 2.5. The Composition Schema for Example 2.11

also be generated by that FSA composition. One interesting question is that given L, is it always possible to synthesize an FSA composition $\mathcal{S}'$ such that $\mathcal{C}(\mathcal{S}') = closure(L)$? The answer is negative. Consider the following example.

Example 2.11 In Fig. 2.5 we present a composition schema that consists of three peers. Consider the language $L = \{ab, bac\}$ (L is described using an FSA in Fig. 2.5). It is obvious that $closure(L) = L$.

Suppose $\mathcal{S}$ is an FSA composition where $closure(L) = \mathcal{C}(\mathcal{S})$. Consider the local run on peer p_2 for the conversation bac. The send of c must be after the send of b, however the consumption of a may be after the send of c. This implies that bac or bca or both must be accepted by p_2 (if we drop the "!" and "?" from p_2 and regard p_2 as a standard FSA which accepts words). Similarly we can infer that ab must be accepted by p_1.

If bac is recognized by p_2, consider the scenario that p_1 takes the local execution path ab and p_2 takes the path bac. It is not hard to see that abc is a conversation, since p_2 sends b while having a in its input queue. Similarly for the case bca is accepted by p_2, we can also show that abc is a conversation. Now conversation abc is not contained in $closure(L)$, and hence $closure(L) \neq \mathcal{C}(\mathcal{S})$. ∎

The following proposition summarizes Example 2.11

Proposition 2.12 There exists a composition schema (P, M) and a language $L \subseteq M^*$ such that for each FSA composition $\mathcal{S}$ over schema (P, M):

$$\mathcal{C}(\mathcal{S}) \neq \mathit{closure}(L).$$

2.6.4 Local Prepone and *ws*-closure

One reason that not every language L closed under *closure* can be a conversation set is that the PREPONE operator (which is applied to global conversations) is too weak. Consider the projection of the conversation *abc* on p_2 in Example 2.11. If it is not accepted by p_2, it must be the result of applying one or more "prepone" like swaps on an accepted word. Since p_2 must accept *bac* or *bca*, as argued in Example 2.11, it is not hard to see that *abc* can be generated from an accepted word of p_2, by doing one swap on b and a for *bac* or applying two swaps (swap a with c and then swap a with b) for *bca*. Note that this type of swap differs from PREPONE since the former is applied locally instead of globally. Secondly, we allow the receiver of the first message and the sender of the second message to be the same, which is forbidden in PREPONE. We call this type of swap a *local prepone*.

Definition 2.7 *Let p_i be a peer of an FSA composition $\mathcal{S}$, a local prepone operator* LP_i *is a function from M_i^* to $2^{M_i^*}$, and for each word w in M_i^* if w can be written as $w = w'm_1m_2w''$, where p_i is the sender of m_1 and the receiver of m_2, then the word $w'm_2m_1w''$ is included* $\text{LP}_i(w)$.

Definition 2.8 *Given a composition schema (P, M) where $n = |P|$, and a language $L \subseteq M^*$. The* ws-closure *of L, written as $wsc(L)$, is defined as follows:*

$$wsc(L) = \text{JOIN}(\text{LP}_1^*(\pi_1(L)), \ldots, \text{LP}_n^*(\pi_n(L))),$$

where LP_i^* represents the reflexive and transitive closure of LP_i, for each peer prototype p_i.

Since local prepone is less restrictive than global prepone, it immediately follows that $closure(L) \subseteq wsc(L)$. Next we are going to present a simple mathematical characterization of the conversation set for any arbitrary FSA composition, using the local prepone and join operators. Before the presentation of this result, we need to study the local execution of each peer during a run.

2.6.5 Local Execution

The definition of $\rightarrow$ given for a web service composition has the effect of *generating* words. Now to study the local execution of each peer, we define a kind of converse for each individual FSA peer which has the effect of *consuming* words. Let $\mathcal{A}_i$ be an FSA peer $(M_i, T_i, s_i, F_i, \delta_i)$ which implements peer prototype p_i. A *local (l-)configuration* of $\mathcal{A}_i$ is a triple $(t, u, v) \in T_i \times (M_i^{in})^* \times M_i^*$. In an l-configuration (t, u, v), t is the current state of the peer $\mathcal{A}_i$, u is the sequence of messages in the input queue of $\mathcal{A}_i$, v is a sequence of "future messages" including the incoming messages not yet in the queue of $\mathcal{A}_i$ and the messages to be sent out by $\mathcal{A}_i$ (i.e., v represents the remaining portion of a conversation projected to the messages visible to $\mathcal{A}_i$).

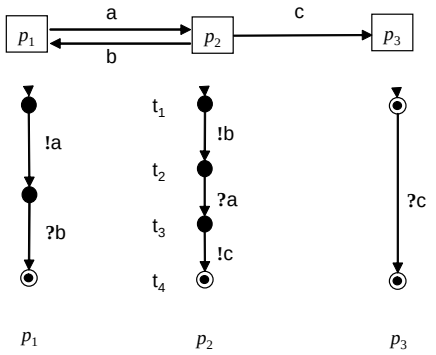


Figure 2.6. The FSA Composition for Example 2.11

We define $(t, u, v) \rightarrow_i (t', u', v')$ for a pair of l-configurations (t, u, v) and (t', u', v') if one of the following holds for some $a \in M_i^{in}$ and some $b \in M_i^{out}$:

- (Consuming a message from the queue) $u = au'$, $v = v'$, and $(t, ?a, t') \in \delta_i$,
- (Sending a message) $u = u'$, $v = bv'$, and $(t, !b, t') \in \delta_i$,
- (ϵ-move) $u = u'$, $v = v'$, and $(t, \epsilon, t') \in \delta_i$, or
- (En-queuing a message) $t = t', u' = ua, v = av'$.

Definition 2.9 *Given an FSA composition $\langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ where $n = |P|$, let $\gamma_i = c_0, c_1, \ldots, c_k$ be a list of l-configurations for peer $\mathcal{A}_i$. γ_i is a* local (l)-run *of $\mathcal{A}_i$ if c_0 can be written as $c_0 = (s_i, \epsilon, w)$ where s_i is the initial state of $\mathcal{A}_i$, $w \in M_i^*$, and $c_a \rightarrow_i c_{a+1}$ for each $0 \leq a < k$. γ_i is a* complete local run *if the following additional condition is satisfied: $c_k = (q, \epsilon, \epsilon)$ where q is a final state of $\mathcal{A}_i$. When γ_i is a complete l-run, we call w, the word in the initial l-configuration of γ_i, a* complete local execution *of $\mathcal{A}_i$.*

Example 2.13 Fig. 2.6 is one FSA composition that generates each word of $L = \{ab, bac\}$ in Example 2.11. Let $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ be the peer implementation

for p_1, p_2, p_3, respectively. Study the conversation abc, its projection to peer prototype p_2, i.e., abc, is a complete local execution of $\mathcal{A}_2$, because there exists a corresponding complete local run of $\mathcal{A}_2$ as follows.

$$(t_1, \epsilon, abc) \rightarrow_2 (t_1, a, bc) \rightarrow_2 (t_2, a, c) \rightarrow_2 (t_3, \epsilon, c) \rightarrow_2 (t_4, \epsilon, \epsilon)$$

Basically, the above local run consists of 4 steps: (1) message a enters the input queue of $\mathcal{A}_2$, (2) message b is sent out by $\mathcal{A}_2$, (3) message a is consumed, and (4) message c is sent out. ∎

Lemma 2.14 Let $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be an FSA composition. Given a word $w \in M^*$, if for each $i \in [1..n]$, $\pi_i(w)$ is a complete local execution of $\mathcal{A}_i$, then w is a conversation of $\mathcal{S}$. The converse is also true.

Proof: In the following, we give the proof for the proposition "if projection of w to each peer is a complete local execution, then w is a conversation". The proof of the converse is trivial.

Let $w = \alpha_1 \cdots \alpha_m$. Since for each $i \in [1..n]$, the projection $\pi_i(w)$ is a complete local execution, there exists a corresponding complete l-run γ_i for $\pi_i(w)$. We show that w is a conversation by constructing a complete run which simulates each γ_i. The construction has $(m + 1)$ *phases*. Phase 0 is the initialization phase where we simulate in the global run the initial ϵ-moves of each p_i until it advances to an l-configuration that is ready to do a send-message action or an enqueue-message action. Then in each phase j, we simulate the transmission of message α_j, where only the sender and receiver of α_j are involved. We start with the sender of α_j. We execute the send-α_j action, and its follow-up actions such as ϵ-moves

and consume-message actions, until we encounter an enqueue-message or send-message action on a message $\alpha_{j'}$ where $j' > j$. Then we turn to the receiver of α_j, execute the enqueue-α_j action and the follow-up actions until an action related to a later message is reached.

To prove the correctness of the above process, we need to show that after each phase the simulation can always continue, and that at the end of the simulation, the global configuration is consistent with the l-configuration of each peer at the end of its local run. They are guaranteed by the following induction assumption: prior to the phase j ($j \in [1..m+1]$) of the simulation, the following statement (denoted as P) is true: for each peer $\mathcal{A}_i$, its complete local run is simulated up to a l-configuration (t_i, w_1, w_2) where $w_2 = \pi_i(\alpha_j, \ldots, \alpha_m)$, and either t_i is a final state or the next action in the local run of $\mathcal{A}_i$ is a send action or an en-queue action. Obviously, P is satisfied at the end of phase 0. When P holds at the beginning of the phase j where $j \in [1..m]$, the simulation at phase j (which simulates the actions of the sender and the receiver of α_j) guarantees that P holds at the end of phase j (i.e., the beginning of phase $j+1$). ∎

2.6.6 Characterize Conversation Set

We now discuss how to use local prepone and join to characterize the conversation set for an arbitrary FSA web service composition.

Lemma 2.15 Let $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be an FSA web service composition. For each $i \in [1..n]$, a word $w \in M_i^*$ is a complete local execution of $\mathcal{A}_i$ if and only if $w \in \text{LP}_i^*(L(\mathcal{A}_i))$.

Proof: We prove by induction (on the number of the times LP_i operator is applied) that $w \in \text{LP}_i^*(L(\mathcal{A}_i))$ is a sufficient condition for w being a complete local execution. It suffices to show that if a word w is contained in $\text{LP}_i(w')$ for some complete local execution w', w is also a complete local execution. The proof of the claim is straightforward, because we can always construct an l-run for w by modifying the l-run of w'.

Next we prove that $w \in \text{LP}_i^*(L(\mathcal{A}_i))$ is a necessary condition. We show that for any complete local execution w, we can always find $w' \in L(\mathcal{A}_i)$ such that $w \in \text{LP}_i^*(w')$, by applying "reverse prepone" procedure finitely many times. We briefly describe the procedure below. Consider the l-run $c_0 \rightarrow_i \cdots \rightarrow_i c_n$ of the complete local execution w. Let $(q_a, u_1, \alpha u_2) \rightarrow_i (q_{a+1}, u_1, u_2)$ be the first send-message action such that input queue is not empty, i.e., $|u_1| > 0$. Since $!\alpha$ is the first such send-message action, w can be written as $w = w_1 u_1 \alpha u_2$, where w_1 includes those eagerly processed messages before the arrival of any message in u_1. Now let $w^1 = w_1 \alpha u_1 u_2$. Obviously, $w \in \text{LP}_i^{|u_1|}(w^1)$. w^1 is also a complete local execution, because we can construct the corresponding local run for w^1 by modifying the local run for w. Repeat the above procedure, until we cannot find a send-message action with a non-empty queue, then we get a list $w^0, w^1, ..., w^k$ where $w^0 = w$, $w^k \in L(\mathcal{A}_i)$, and for each $0 \leq j < k$, $w^j \in \text{LP}_i^*(w^{j+1})$. Note that w^k is an accepted word of $\mathcal{A}_i$ because all messages are consumed eagerly during the l-run for w^k, i.e., whenever a message is sent out by $\mathcal{A}_i$, its input queue is empty. Hence we have $w \in \text{LP}_i^*(L(\mathcal{A}_i))$. ∎

Lemma 2.14 and Lemma 2.15 immediately imply the following theorem.

Theorem 2.16 Given an FSA composition $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, the conversation set generated by $\mathcal{S}$ can be characterized using the following formula:

$$\mathcal{C}(\mathcal{S}) = \text{JOIN}(\text{LP}_1^*(L(\mathcal{A}_1)), \ldots, \text{LP}_n^*(L(\mathcal{A}_n)))$$

2.6.7 Regular Core

Now consider the following problem: given an FSA composition $\mathcal{S}$, can we find a regular language L as its core such that $wsc(L) = \mathcal{C}(\mathcal{S})$? The following example provides a negative answer.

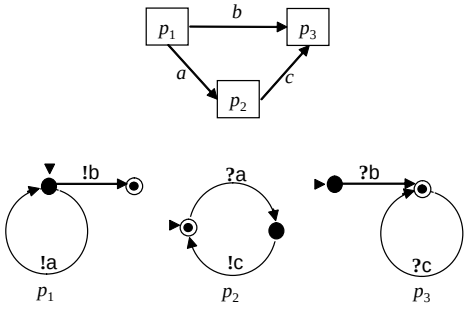


Figure 2.7. The FSA Composition for Example 2.17

Example 2.17 Consider the FSA composition shown in Fig. 2.7, which consists of 3 peers, p_1, p_2, p_3. Intuitively, p_1 sends, say i messages of class a, to p_2, a message b to p_3, and then halt; p_2 responds to each a message by sending one c message to p_3; p_3 expects b at the beginning and then consumes all c messages. It is not hard to see that the only way for p_3 to halt is for p_2 to keep all a messages in its queue till after p_1 sends b to p_3. Thus $L = \{a^i b c^i \mid i \geq 0\}$ is its conversation set.

Notice that none of the message pairs (a, b), (b, c), and (a, c) can satisfy the condition to apply global prepone or local prepone operators. In addition, the

projection of a word $a^i bc^i$ ($i \geq 0$) to p_2 is $a^i c^i$, which requires the matching of the number of a's and c's. This implies that we cannot construct any new word from the projections of any three different words $a^i bc^i$, $a^j bc^j$, and $a^k bc^k$ where i, j, k are pairwise unequal. The above observation immediately leads to the following property for each subset L' of L:

$$L' = closure(L') = wsc(L')$$

This implies that the conversation set of Fig. 2.7 does not have a regular core. ∎

We summarize Example 2.17 as follows.

Proposition 2.18 There exists an FSA web service composition $\mathcal{S}$ such that $\mathcal{C}(\mathcal{S}) \neq wsc(L)$ for each regular language L.

2.7 Topdown Approach: Conversation Protocols

Proposition 2.18 suggests that adding asynchronous communication significantly increases the power of finite state machines. This unsettling fact motivates us to look for an essentially "weaker" mechanism to describe web service compositions. As attention has to be given on the global behaviors of web service compositions, and LTL properties are defined on conversations. It may be "cheaper" and more direct to provide a specification of the global behaviors, and leave the specification of peers blank.

Definition 2.10 *Let $S = (P, M)$ be a composition schema. An* FSA conversation protocol *over S is a tuple $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ where $\mathcal{A}$ is a finite state*

automaton on alphabet M. We let $L(\mathcal{P}) = L(\mathcal{A})$, i.e., the language recognized by $\mathcal{A}$.

Definition 2.11 *Let $S = (P, M)$ be a composition schema, and let FSA conversation protocol $\mathcal{P}$ and FSA composition $\mathcal{S}$ be both over schema S. We say $\mathcal{S}$* realizes *$\mathcal{P}$ if $\mathcal{C}(\mathcal{S}) = L(\mathcal{P})$. An FSA conversation protocol $\mathcal{P}$ is* realizable *if there exists an FSA composition that realizes $\mathcal{P}$.*

We are interested in the following question: given an FSA conversation protocol $\mathcal{P}$, is it always possible to construct an FSA composition that realizes $\mathcal{P}$? In the following we show the answer is negative.

Example 2.19 Let (P, M) be a composition schema which consists of four peers p_1, p_2, p_3 and p_4. Its alphabet M consists of two messages a which is from p_1 to p_2 and b which is from p_3 to p_4. Suppose $\mathcal{P}$ is an FSA conversation protocol over (P, M) where $L(\mathcal{P}) = \{ab\}$. It is clear that any peer implementation which generates conversation ab can also generate ba as well, because there is no way to let p_3 and p_1 synchronize their send operation. Hence the conversation protocol $\mathcal{P}$ is not realizable. ∎

The above example can be summarized using the following proposition.

Proposition 2.20 There exist FSA conversation protocols that are not realizable.

Note that although an FSA conversation protocol may be not realizable, we can always find an FSA composition $\mathcal{S}'$ such that the protocol is the "regular core" of $\mathcal{C}(\mathcal{S}')$.

Proposition 2.21 For any FSA conversation protocol $\mathcal{P}$ which is over some composition schema S that has n peer prototypes, there is an FSA composition $\mathcal{S}$ over S such that $\mathcal{P}$ is the regular core of $\mathcal{S}$, i.e.,

$$\mathcal{C}(\mathcal{S}) = wsc(L(\mathcal{P}))$$

Proof: Let $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$. We can construct the FSA web service composition $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ as follows. Each peer implementation $\mathcal{A}_i$ is essentially the projection of $\mathcal{A}$ to peer prototype p_i: replace all edges in $\mathcal{A}$ that are irrelevant to p_i by ϵ moves, change edges of messages sent to p_i as receive-transitions, and change edges of messages sent by p_i as send-transitions. It is clear that $L(\mathcal{A}_i) = \pi_i(L(\mathcal{A}))$ for each peer prototype p_i. Then by Theorem 2.16, we have $\mathcal{C}(\mathcal{S}) = \textsc{join}(\textsc{lp}_1^*(L(\mathcal{A}_1)), \ldots, \textsc{lp}_n^*(L(\mathcal{A}_n)))$, hence $\mathcal{C}(\mathcal{S}) = wsc(L(\mathcal{P}))$. ∎

2.8 Modeling of Reactive Web Services

This section studies a variation of our model [39] to specify reactive web services, which may have infinitely long interactions. We use Büchi automata [15] to specify such services. Different from a standard FSA, a Büchi automaton's acceptance condition requires that some final state in the automaton is visited infinitely often during the run for an accepted word. Note that in our Büchi automaton specification for a peer, there may exist ϵ transitions, hence a Büchi peer (as an automaton to recognize words) can accept both finite and infinite words. Given an alphabet Γ, let Γ^ω denote the set of infinite words on Γ, $\Gamma^{\leq\omega} = \Gamma^* \cup \Gamma^\omega$. Formally, a reactive web service composition is defined as follows.

Definition 2.12 *A* Büchi web service composition *(or "reactive composition")*

is a tuple $\langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ *where* $n = |P|$*, and* (P, M) *is the composition schema. For each* $i \in [1..n]$*, peer implementation* $\mathcal{A}_i$ *is a Büchi automaton represented as the tuple* $(M_i, T_i, s_i, F_i, \delta_i)$ *where* M_i*,* T_i*,* s_i*,* F_i*, and* δ_i *are the alphabet, set of states, initial state, final states and transition relation, respectively. A transition can be one of the three types:* $(t_1, !a, t_2)$*,* $(t_1, ?b, t_2)$*, and* (t_1, ϵ, t_2)*, where* $t_1, t_2 \in T_i$*,* $a \in M_i^{out}$*, and* $b \in M_i^{in}$*. A word* $w \in M_i^{\leq \omega}$ *is accepted by* $\mathcal{A}_i$ *if there exists a corresponding run for* w *such that a final state is visited infinitely many times.*

In the following, we briefly present the technical results on the reactive (Büchi) composition model. Since most results are similar to those of the non-reactive (FSA) composition model, we only give the detail proof for the different parts between the two models. First, we have to redefine the notions of complete run and conversation. Note that the definition of a global configuration is the same for both the Büchi and the FSA composition models.

Definition 2.13 *Let* $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ *be a Büchi composition, and let* $\gamma = c_0 c_1 c_2 \cdots$ *be a finite or infinite sequence of configurations.* γ *is a* run *if it satisfies the first two of the following four conditions.* γ *is a* complete run *if it is an infinite configuration sequence that satisfies all of the four conditions.*

1. $c_0 = (\epsilon, s_1, \ldots, \epsilon, s_n, \epsilon)$ *(*s_i *is the initial state of* p_i *for each* $i \in [1..n]$*),*
2. *for each* $0 \leq i < |\gamma| - 1$*,* $c_i \rightarrow c_{i+1}$*,*
3. *for each* $\ell \in [1..n]$ *and each* $i \geq 0$*, there exist* $j > i$ *and* $k > i$ *such that*
   (a) t_ℓ^j *is a final state, where* t_ℓ^j *is the state of* p_ℓ *in* c_j*, and*

(b) either Q_ℓ^k *is empty or* $head(Q_\ell^k) \neq head(Q_\ell^i)$ *if* $Q_\ell^i \neq \epsilon$*, where* Q_ℓ^i *and* Q_ℓ^k *are the queue contents of* p_ℓ *in* c_i *and* c_k *respectively.*

4. for each $i \geq 0$*, there exists* $j > i$ *such that* $gw(c_i) \neq gw(c_j)$*.*

An infinite word $w \in \Sigma^\omega$ *is a* conversation *of* $\mathcal{S}$ *if there exists a complete run* $c_0 c_1 c_2 \cdots$ *of* $\mathcal{S}$ *such that for each* $i \geq 0$*,* $gw(c_i)$ *is a finite prefix of* w*. Let* $\mathcal{C}(\mathcal{S})$ *denote the set of conversations of* $\mathcal{S}$*.*

In Definition 2.5, Condition (3) requires that during a complete run the Büchi acceptance condition of each peer should be met, and all messages stored in input queues should be eventually consumed; Condition (4) specifies that global message exchange should always eventually advance. During a complete run, a peer is said to *terminate* at some configuration c_i if after c_i the peer takes ϵ transitions only and some final state is visited infinitely often. The notions of "completable run", "stuck", and "receptive" are the same as the FSA composition model.

2.8.1 Conversation Set Is Not ω-Regular

Consider the Büchi composition in Fig. 2.8, which consists of three peers: an Investor, an Online Stock Broker, and a Research Department. In each round of message exchange, Online Stock Broker sends a list of **R**awData to Research Department for further analysis, where for each **R**awData one **D**ata is generated and sent to Investor. Messages **E**ndofRdata, **S**tart, and **C**omplete are intended to synchronize the three peers. Finally, Investor acknowledges Online Stock Broker with **A**ck so that a new round of data processing can start. This seemingly simple

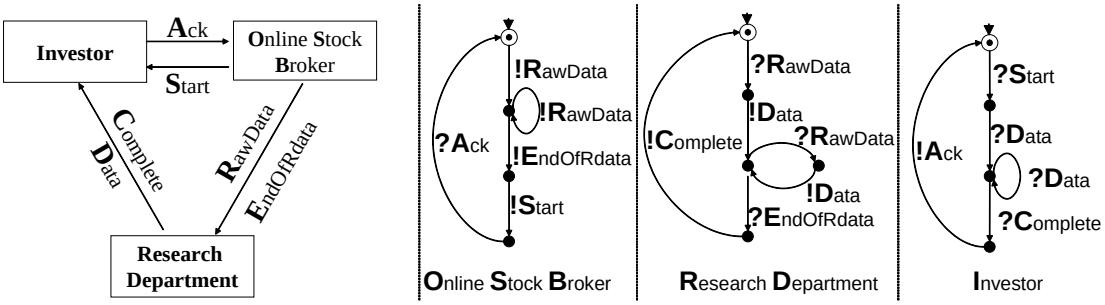


Figure 2.8. Fresh Market Update Service

example produces a non ω-regular set of conversations. Consider its intersection with an ω-regular language $(\mathbf{R}^*\mathbf{ESD}^*\mathbf{CA})^\omega$ where each message is represented by its initial capital letter. It is easy to infer that the result is $(\mathbf{R}^i\mathbf{ESD}^i\mathbf{CA})^\omega$, because each **Data** sent by Research Department should match a **RawData**, and note that during each round all **RawData** are stored in the queue of Research Department before the first **Data** is sent. In addition, **Start** should arrive earlier than **Data**, otherwise Investor gets stuck. Because the number of **R**'s and **D**'s must be equal in the intersection, by an argument similar to pumping lemma, the conversation set can not be recognized by any Büchi automata. The following proposition summarizes the above discussion.

Proposition 2.22 There exists a Büchi composition $\mathcal{S}$ such that $\mathcal{C}(\mathcal{S})$ is not accepted by any Büchi automaton.

2.8.2 LTL Model Checking

The LTL properties defined on infinite conversations are slightly different from the FSA model. Given LTL formulas ϕ, and φ, an atomic proposition $\psi \in AP$, and a word $w \in M^\omega$, the syntax and semantics of LTL formula can be defined as

follows:

$$
\begin{array}{lll}
w \models \psi & \text{iff} & w_0 \in \psi \\
w \models \neg\phi & \text{iff} & w \not\models \phi \\
w \models \phi \wedge \varphi & \text{iff} & w \models \phi \text{ and } w \models \varphi \\
w \models \phi \vee \varphi & \text{iff} & w \models \phi \text{ or } w \models \varphi \\
w \models \mathbf{X}\phi & \text{iff} & w^1 \models \phi \\
w \models \mathbf{G}\phi & \text{iff} & \text{for all } i \geq 0, w^i \models \phi \\
w \models \phi \mathbf{U} \varphi & \text{iff} & \text{there exists } j \geq 0 \text{ such that } w^j \models \varphi \text{ and,} \\
 & & \text{for all } 0 \leq i < j, w^i \models \phi \\
w \models \mathbf{F}\phi & \text{iff} & w \models M\mathbf{U}\phi
\end{array}
$$

Similarly, a Büchi composition $\mathcal{S}$ satisfies an LTL property ϕ if and only if each conversation w in $\mathcal{C}(\mathcal{S})$ satisfies ϕ. Following the same argument for the FSA composition model, we have the undecidability result of the LTL model checking for the reactive model.

Theorem 2.23 Given a Büchi composition $\mathcal{S}$ and an LTL property ϕ, determining if $\mathcal{S} \models \phi$ is undecidable.

2.8.3 Characterization of Conversation Sets

We now discuss how to characterize the conversation set for an arbitrary Büchi composition. The idea is similar to that of the FSA composition model, however, the definition of complete local execution and local prepone have to be redefined.

Definition 2.14 *Given a Büchi composition* $\langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ *where* $n = |P|$, *a* local (l-)configuration *of* $\mathcal{A}_i$ *is a triple* $(t, u, v) \in T_i \times (M_i^{in})^* \times M_i^{\leq \omega}$. *Let*

$\gamma_i = c_0, c_1, \ldots$ *be an infinite list of l-configurations for peer* $\mathcal{A}_i$. γ_i *is a* complete local (l)-run *of* $\mathcal{A}_i$ *if the following conditions are satisfied by* γ_i*:*

1. *Let* c_0 *be written as* $c_0 = (s_i, \epsilon, w)$. s_i *is the initial state of* $\mathcal{A}_i$*, and*
2. $c_j \rightarrow_i c_{j+1}$ *for each* $j \geq 0$*, and*
3. *for each* $j \geq 0$*, there exists* $k > j$ *such that* c_k *can be written as* (t, u, v) *where* t *is a final state of* $\mathcal{A}_i$*, and*
4. *for each* $j \geq 0$*, there exists* $k > j$ *such that if* $c_j = (t, u, v)$ *and* $c_k = (t', u', v')$*, then* $u' = \epsilon$ *or* $u_0 \neq u'_0$ *(*u_0 *and* u'_0 *are the first message of* u *and* u'*, respectively).*

The word $w \in M^{\leq\omega}$*, which appears in the initial l-configuration of* γ_i*, is called a* complete execution *of* $\mathcal{A}_i$.

Lemma 2.24 Let $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be a Büchi composition. Given a word $w \in M^{\omega}$,[3] if for each $i \in [1..n]$, $\pi_i(w)$ is a complete local execution of $\mathcal{A}_i$, then w is a conversation of $\mathcal{S}$. The converse is also true.

Proof: The proof is almost the same as the proof for Lemma 2.14: construct the global run to simulate each local run for the projection of the word. The induction in the proof of Lemma 2.14, which shows that the simulation process can always proceed, works for the reactive model too, because a message cannot stay in queue for an infinite long time period (as required in Definition 2.13 and Definition 2.14). ∎

[3]Note that the use of M^{ω} (instead of $M^{\leq\omega}$) ensures the satisfaction of the condition 4 in Definition 2.13

We now define a *swap closure* operator which is essentially the "infinite" reflexive and transitive closure of local prepone operators for infinite words.

Let $p_i = (M_i^{in}, M_i^{out})$ be a peer prototype. Given a word $w \in M_i^{\leq\omega}$, the *swap closure* of w at p_i, written as $\text{SC}_i(w)$, is a subset of $2^{M_i^{\leq\omega}}$ which includes every word w' satisfying the following conditions:

1. $\pi_{M_i^{in}}(w') = \pi_{M_i^{in}}(w)$, and
2. $\pi_{M_i^{out}}(w') = \pi_{M_i^{out}}(w)$, and
3. for each integer $j \geq 0$, the number of input messages (of p_i) in the prefix $w_0 \dots w_j$ of w is no greater than that of the prefix $w'_0 \dots w'_j$ of w'. (w_j and w'_j are the j-th message in w and w', respectively.)

Lemma 2.25 Let $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \dots, \mathcal{A}_n \rangle$ be a Büchi composition. For each $i \in [1..n]$, a word $w \in M_i^{\leq\omega}$ is a complete execution of $\mathcal{A}_i$ if and only if $w \in \text{SC}_i(L(\mathcal{A}_i))$.

Proof: It suffices to show that for each word $w \in M_i^{\leq\omega}$, and its corresponding l-run γ_i: if $w' \in \text{SC}_i(w)$, we can construct a corresponding l-run γ'_i for w' from γ_i. The l-run γ'_i can be defined as an reordering of γ_i such that when excluding the enqueue-actions from the two l-runs, the sequence of ϵ-moves, send-actions and consume-actions is the same for γ_i and γ'_i. The only difference is that the enqueue-actions of γ'_i are "preponed" forward, which, still allows the γ'_i to proceed following the logic (on send- and consume-actions) of γ_i, because for a consume-action in γ_i, its corresponding enqueue-action in γ'_i can always happen earlier. ∎

Lemma 2.24 and Lemma 2.25 immediately lead to the following theorem.

Theorem 2.26 For each Büchi composition $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, its conversation set $\mathcal{C}(\mathcal{S}) = \textsc{join}(\textsc{sc}_1(\mathcal{L}(\mathcal{A}_1)), \ldots, \textsc{sc}_n(\mathcal{L}(\mathcal{A}_n)))$.

2.8.4 Büchi Conversation Protocols

We can also specify reactive compositions in a top-down fashion, where a Büchi automaton specifies the desired set of conversations. We call a Büchi automaton *nonredundant* if for each of its states there is an accepted word whose run traverses through the state.

Definition 2.15 *Given a composition schema (P, M), a* Büchi (or reactive) conversation protocol $\mathcal{P}$ *is a tuple $\langle (P, M), \mathcal{A} \rangle$. $\mathcal{A}$ is a nonredundant Büchi automaton (without ϵ-transitions) on alphabet M. The conversation set defined by $\mathcal{P}$ is the language accepted by $\mathcal{A}$, i.e., $L(\mathcal{P}) = L(\mathcal{A})$.*

Similar to FSA conversation protocols, we can define the realizability of a Büchi conversation protocol.

Definition 2.16 *Let $\mathcal{P}$ and $\mathcal{S}$ be a Büchi conversation protocol and a Büchi web service composition over a composition schema S, respectively. We say $\mathcal{S}$* realizes *$\mathcal{P}$ if $\mathcal{C}(\mathcal{S}) = L(\mathcal{P})$. A Büchi conversation protocol is* realizable *if there exists a Büchi composition that realizes it.*

The projection of a Büchi conversation protocol to its peers is similar to that of a FSA conversation protocol. Given a Büchi automaton $\mathcal{A}$ over a composition

schema $S = (P, M)$, and a peer prototype $p_i \in P$, the *projection* of $\mathcal{A}$ onto p_i, written as $\pi_i(\mathcal{A})$, is a Büchi automaton $\mathcal{A}_i$ obtained from $\mathcal{A}$ by replacing each move for a message not in the alphabet of p_i by an ϵ-move. Note that, due to the existence of ϵ-move, $L(\pi_i(\mathcal{A}))$ may contain words of finite length.

2.9 Related Work

In approaches such as CSP [49], I/O automata [57] and interface automata [3], the communicating processes execute a send and a corresponding receive action synchronously, while in our model, messages are stored in FIFO buffer first. Our model of web service compositions is slightly different than the Communicating Finite State Machines (CFSM) model in [13], and almost identical to its variation named Single-Link Communicating Finite State Machines (SLCFSM) [69], except that both Büchi automata and standard FSA are used in our model. In SLCFSM and our model, messages are exchanged through a virtual common medium and stored in the queue associated with the receiver, whereas in [13] each pair of communicating machines use isolated communication channels and each channel has its own queue. The idea of using CFSM with FIFO queues to capture indefinite delay of messages (signals) is similar to many other published models like Codesign Finite State Machine [20], and Kahn Process Networks [53]. Other formalisms like π-Calculus [61] and the recent Microsoft Behave! Project [72] are used to describe concurrent, mobile and asynchronously communicating processes. Finally, [10] studies "quasi-realtime" automata with queues. These are single automata with one or more queues, where an automaton can write a bounded number of letters on the queue(s) for each input letter read. In [10], the

input and queue alphabets may be different; in our framework the alphabets are identical.

Brand and Zafiropulo have shown in [13] that CFSM with perfect FIFO queues are as powerful as Turing Machines. Thus it is not hard to infer that LTL model checking on our web service composition model is undecidable. This undecidability result is caused by the unbounded FIFO queues, and in [33], many problems are proved to be undecidable even for two identical communicating processes. The transaction sequential consistency problem in [8] provides another perspective for understanding the queue effect, where independent transactions are allowed to commute (which resembles the prepone operator). Although FIFO is the most popular assumption about network environment for web services, industry messaging platforms can provide services with different qualities. For example, Java Message Service [52] allows users to tune the priority, expiration date, and persistence of messages to deliver. Undoubtedly, different communication assumptions lead to different analysis complexity for communicating systems. For example, in [2], it is shown that, if perfect FIFO channels are replaced by *lossy* channels, many problems of analyzing CFSM become decidable. As one of our future research plans, it is interesting to study the variations of our current framework with different network assumptions (e.g. FIFO, non-lossy but reordering, and lossy message delivery).

While most industry solutions (e.g. BPEL4WS and WSCI) favor bottom-up specifications, there are efforts to specify distributed systems in a top-down fashion, for example the IBM conversation support project [51] and the Message Sequence Chart model [63]. The notion of conversation protocol resembles those industry initiations, however, there are still interesting differences. In the fol-

lowing, we give a comparative study on the expressive power of our conversation oriented model and the MSC graph model [7].

2.9.1 Comparison with Message Sequence Charts

In this section, we discuss the expressive power of three specification approaches for a composition of web services: the top-down FSA conversation protocol, the bottom-up FSA composition, and the MSC graph model.

MSC Graphs

MSC [63] is a widely used scenario specification approach for concurrent systems. An MSC consists of a finite set of peers, where each peer has one single sequence of send/receive events of messages (e.g. peer B in Fig. 2.9(a) has the sequence of $?\alpha\ !\beta$). We call that sequence the *event order* of that peer. There is a bijective mapping (represented using arrows in Fig. 2.9) that matches each pair of send and receive events. Given an MSC M, its language $L(M)$ is the set of *linearizations* of all events that follow the event order at each peer. Essentially $L(M)$ captures the "join" of local views from each peer. A formal definition of MSC can be found in [7].

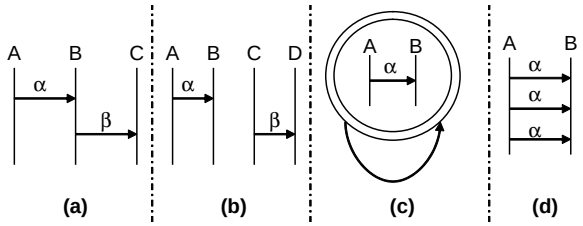


Figure 2.9. MSC Example

Example 2.27 Fig. 2.9 (a) and (b) are two MSCs, and let them be M_a and M_b respectively. Obviously $L(M_a) = \{!\alpha\ ?\alpha\ !\beta\ ?\beta\}$, and $L(M_b) = \{\ !\alpha\ ?\alpha\ !\beta\ ?\beta,\ !\alpha\ !\beta\ ?\alpha\ ?\beta,\ \ !\alpha\ !\beta\ ?\beta\ ?\alpha,\ \ !\beta\ ?\beta\ !\alpha\ ?\alpha,\ \ !\beta\ !\alpha\ ?\beta\ ?\alpha,\ \ !\beta\ !\alpha\ ?\alpha\ ?\beta\ \}$. ∎

An MSC Graph [7] is a finite state automaton, and each node of the graph is associated with an MSC. Given an MSC graph G, a word w is accepted by G, if and only if there exists an accepted path of G, and w is a linearization of the MSC that is the result of concatenating MSCs along that path.

Example 2.28 Fig. 2.9(c) presents an MSC graph G which consists of a single state. A linearization $!\alpha\ !\alpha\ ?\alpha\ ?\alpha\ !\alpha\ ?\alpha$ belongs to $L(G)$ because if we run G by traversing the transition twice, we shall connect the MSC associated with the state three times, and the resulting MSC is Fig. 2.9(d). Obviously $!\alpha\ !\alpha\ ?\alpha\ ?\alpha\ !\alpha\ ?\alpha$ is one of the linearizations of Fig. 2.9(d). Actually the language $L(G)$ can be described using the following [7]:

$$L(G) = \{(!\alpha \mid ?\alpha)^* \mid |!\alpha| = |?\alpha| \wedge \text{ for any prefix } |!\alpha| \geq |?\alpha|\}.$$

Note that semantics of MSC graph is not the concatenation of the language of the MSCs of each passed states. ∎

We now present some properties of MSC graphs, which will be useful in the later comparison of expressive power. Given a linearization ℓ that consists of send and receive events, $\pi_{send}(\ell)$ returns a sequence of send events by deleting receive events in ℓ. Since $\pi_{send}(\ell)$ contains send events only, we remove "!" for each event in $\pi_{send}(\ell)$ for simplicity.

Lemma 2.29 Given an MSC M, we can always construct an FSA $\mathcal{A}$ such that $L(\mathcal{A}) = \pi_{send}(L(M))$.

Proof: Enumerate and collect the $\pi_{send}(\ell)$ of each linearization ℓ of M. Obviously, the set is a finite set of words, which can be recognized by an FSA. ∎

Based on Lemma 2.29, and the following restrictive join closure operator, we are able to construct an FSA for each MSC graph.

Definition 2.17 *Given a composition schema (P, M) where $n = |P|$, and a language $L \subseteq M^*$, the* restrict join closure *of L, written as* $\textsc{rjoinc}(L)$ *is defined as follows:*

$$\textsc{rjoinc}(L) = \{w \mid \textit{there exists a word } w' \in M^* s.t. \forall i \in [1..n] : \pi_i(w) = \pi_i(w')\}$$

Obviously, for any language L, $\textsc{rjoinc}(L) \subseteq \textsc{joinc}(L)$. With RJOINC, we have the following lemma.

Lemma 2.30 Given two MSCs M_1 and M_2 on a same composition schema (P, M) where $n = |P|$, and let $\mathcal{A}_1$ and $\mathcal{A}_2$ be the corresponding FSA that specifies their projection on send events respectively, then the following is true:

$$\pi_{send}(L(M_1 \circ M_2)) = \textsc{rjoinc}(L(\mathcal{A}_1 \circ \mathcal{A}_2))$$

Here "$\circ$" is the concatenation operator. $M_1 \circ M_2$ is a new MSC graph by concatenating the event order for each peer prototype, and $\mathcal{A}_1 \circ \mathcal{A}_2$ is a new FSA constructed from $\mathcal{A}_1$ and $\mathcal{A}_2$ by linking each final state of $\mathcal{A}_1$ to the initial state of $\mathcal{A}_2$ with ϵ-transition, and making each state which is originally a part of $\mathcal{A}_1$ a non-final state.

Proof: For each peer prototype p_i, let $\ell_{i,1}$ and $\ell_{i,2}$ denote the event order of p_i in M_1 and M_2 respectively. For each linearization ℓ of $M_1 \circ M_2$, we have

$$\forall i \in [1..n] : \pi_i(\ell) \;=\; \ell_{i,1} \;\circ\; \ell_{i,2} \tag{2.3}$$

Now for each word $w \in L(\mathcal{A}_1 \circ \mathcal{A}_2)$, w can be written as $w = w_1 \circ w_2$ where $w_1 \in L(\mathcal{A}_1)$ and $w_2 \in L(\mathcal{A}_2)$. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are the corresponding FSA for M_1 and M_2 respectively, for each $i \in [1..n]$: $\pi_i(w) = \pi_{send}(\ell_{i,1})$, and $\pi_i(w) = \pi_{send}(\ell_{i,1})$. Combined with Equation 2.3, this immediately leads to the following:

$$\forall i \in [1..n]\ \forall \ell \in L(M_1 \circ M_2)\ \forall w \in L(\mathcal{A}_1 \circ \mathcal{A}_2) : \pi_i(\pi_{send}(\ell)) = \pi_i(w) \tag{2.4}$$

Then $\pi_{send}(L(M_1 \circ M_2)) = \textsc{rjoinc}(L(\mathcal{A}_1 \circ \mathcal{A}_2))$ can be inferred from Equation 2.4 directly. ∎

Lemma 2.31 For each MSC graph G there is a corresponding FSA $\mathcal{A}$ such that $\pi_{send}(L(G)) = \textsc{rjoinc}(L(\mathcal{A}))$.

Proof: Now given an MSC graph G, we can always construct an FSA $\mathcal{A}$ for G as follows: replace each MSC in each state of G with the corresponding FSA as shown in Lemma 2.29, properly connect the final states and initial states of neighboring FSAs by ϵ transitions. Now to prove that the resulting FSA does capture the projection of the linearization set to send-events, we need a strengthened version of Lemma 2.30: given k MSCs $M_1, M_2, \ldots, M_k$ and their corresponding FSA $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_k$, then $\pi_{send}(L(M_1 \circ M_2 \circ \cdots \circ M_k)) = \textsc{rjoinc}(L(\mathcal{A}_1 \circ \mathcal{A}_2 \circ \cdots \circ \mathcal{A}_k))$. Its proof is straightforward, based on the proof of Lemma 2.30. ∎

Expressive Power

At the first glance, the difference between the MSC graph framework and the conversation oriented framework is trivial – the MSC model captures receive events while the conversation oriented model does not. This different modeling perspective is caused by the different application domain of the two models. In web service composition, we care only about "observable behaviors", while message consumption is regarded as internal operations inside each peer. The seemingly trivial difference, however, leads to interesting and significant differences in the technical results of realizability analysis in Chapter 3. But first let us study the relative expressive power of the two models.

MSC graph vs. Conversation Protocol:

Example 2.32 Let M_a and M_b be the two MSCs in Fig. 2.10. Obviously $\pi_{send}(L(M_a)) = \pi_{send}(L(M_b)) = \{\alpha\beta, \beta\alpha\}$. But $L(M_a) \neq L(M_b)$ because in M_a the $?\alpha$ precedes the $?\beta$, while in M_b the $?\beta$ is before $?\alpha$. ∎

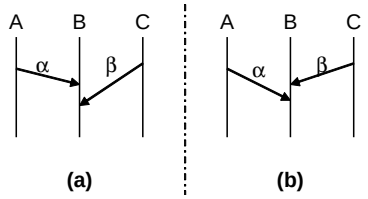


Figure 2.10. Two MSC examples

Example 2.32 can be summarized as follows.

Proposition 2.33 There exist two MSCs M_1 and M_2, and one conversation protocol $\mathcal{P}$ where $L(M_1) \neq L(M_2)$ however $\pi_{send}(L(M_1)) = \pi_{send}(L(M_2)) = L(\mathcal{P})$.

Consider the conversation protocol which specifies one single conversation $\alpha_{A\to B}\ \gamma_{C\to A}$. No MSC graph can capture the scenario described by this conversation protocol, because there is no way to enforce the send of α to precede γ. Hence we have the following proposition.

Proposition 2.34 There exists a conversation protocol $\mathcal{P}$ where for any MSC graph G, the following is true: $\pi_{send}(L(G)) \neq L(\mathcal{A})$.

Generally the above two Propositions imply that conversation protocol and MSC graph are incomparable with regard to expressive power: there are two different MSC graphs where conversation protocols cannot distinguish the difference; whereas there are conversation protocols which have no equivalent MSC graphs.

MSC Graph vs. Bottom-up FSA compositions:

Now we show that the MSC Graph model and the bottom-up specified FSA composition model are incomparable concerning the expressive power.

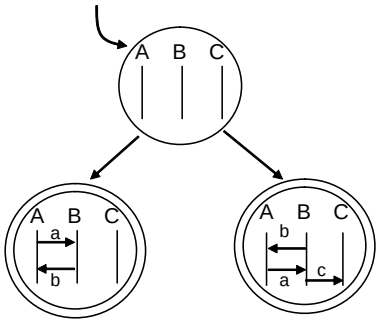


Figure 2.11. The MSC Graph for Proposition 2.35

Proposition 2.35 There exists an MSC graph G such that for any FSA composition $\mathcal{S}$, $\mathcal{C}(\mathcal{S}) \neq \pi_{send}(G)$.

Proof: Let G be the MSC graph presented in Fig. 2.11. Obviously, $\pi_{send}(L(G)) = \{ab, bac\}$. Recall that $\{ab, bac\}$ is the language discussed in Example 2.11, any FSA composition that generates ab and bac will generate abc as well. Hence, there does not exist an FSA composition $\mathcal{S}$ such that $\mathcal{C}(\mathcal{S}) = \pi_{send}(L(G))$. ∎

A converse question is: given a bottom-up web service composition S, can its conversation always be captured by an MSC graph? The following proposition gives a negative answer.

Proposition 2.36 There exist an FSA composition $\mathcal{S}$ such that for each MSC Graph G: $\mathcal{C}(S) \neq \pi_{send}(L(G))$.

Proof: We have shown that the FSA composition in Fig. 2.7 produces the conversation set $\{a^n bc^n \mid n > 0\}$. Now suppose that there is an MSC graph G such that $\pi_{send}(L(G)) = \{a^n bc^n \mid n > 0\}$. By Lemma 2.31, there exists an FSA $\mathcal{A}$ such that $\text{RJOINC}(L(\mathcal{A})) = \{a^n bc^n \mid n > 0\}$. However it is shown in Example 2.17 that for each subset L' of $\{a^n bc^n \mid n > 0\}$, $wsc(L') = L'$, which follows that $\text{RJOINC}(L') = L'$. Then we get $L(\mathcal{A}) = \text{RJOINC}(L(\mathcal{A})) = wsc(L(\mathcal{A})) = \{a^n bc^n \mid n > 0\}$, and this contradicts with the fact that $\mathcal{A}$ is a finite state automaton. ∎

Chapter 3

Realizability and Synchronizability Analyses

It is shown in Chapter 2 that not every conversation protocol is realizable, i.e., given an arbitrary conversation protocol there may not exist web service compositions that can realize it. On the other hand, the general problem of deciding if a web service composition satisfies an LTL property is undecidable. In addition, even if we can prove that the conversation set of a web service composition satisfies a certain LTL property, designers still have no absolute confidence, because the conversation set may not capture all possible behaviors of the web service composition (due to the possible existence of deadlock and unspecified message reception). This chapter provides a solution to solve all the above problems.

In [39], we propose several sufficient conditions to restrict the control flows of a Büchi conversation protocol. When these conditions are satisfied, it is possible to

implement the protocol using a set of finite state peers. In particular, the synthesized peers will conform to the protocol by generating only those conversations specified by the protocol. Our results enable a top-down verification strategy where (1) A conversation protocol is specified by a realizable Büchi automaton; (2) The properties of the protocol are verified on the realizable conversation protocol; and (3) The peer implementations are synthesized from the protocol via projection.

Interestingly, for non-reactive FSA conversation protocols, we can achieve better results than those on Büchi conversation protocols. Additional good properties, e.g., freedom of unspecified message reception and freedom of dead-lock, can be guaranteed for the synthesized peers from an FSA conversation protocol. The difference between the two models results from the inequivalence of nondeterministic and deterministic Büchi automata.

Following the idea of realizability analysis, we derive a similar verification strategy called "synchronizability analysis" [40] for bottom-up specified web service compositions. When a set of synchronizability conditions are satisfied, an FSA web service composition will generate the same set of conversations under both the asynchronous and the synchronous communication semantics. LTL verification can be conducted using the synchronous communication semantics, and the verification results hold for the usual asynchronous communication semantics. The synchronizability analysis can be extended to the Büchi web service compositions as well.

Since we have to cover both the FSA and the Büchi models, this chapter is organized as follows. We first introduce the realizability analysis for Büchi

conversation protocols, then additional analysis results are presented for FSA conversation protocols. Finally we introduce the synchronizability analysis for FSA web service compositions, and briefly summarize the similar results on the Büchi model.

3.1 Realizability Analysis for Büchi Conversation Protocols

This section presents the realizability analysis for Büchi conversation protocols. We first pay a short revisit to the realizability problem, and present a powerful weapon to prove a conversation protocol cannot be realizable. Then we introduce three sufficient conditions that can guarantee realizability. Finally, we discuss how "restrictive" these conditions are in real-world applications.

3.1.1 Revisit Realizability

From a Büchi conversation protocol, we can always synthesize a Büchi web service composition by projecting the protocol to each peer prototype. Formally, the projected composition is defined as follows.

Definition 3.1 *Let* $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ *be a Büchi conversation protocol where* $n = |P|$. *The* projected composition of $\mathcal{P}$, *written as* $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$, *is a Büchi web service composition* $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ *such that for each* $1 \leq i \leq n$: $\mathcal{A}_i = \pi_i(\mathcal{A})$.

Next we show that a Büchi conversation protocol is realizable if and only if it can be realized by its projected composition.

Theorem 3.1 A Büchi conversation protocol $\mathcal{P}$ is realizable if and only if it is realized by $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ (i.e., $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}})$).

Proof: Since $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}})$ directly leads to the realizability of $\mathcal{P}$, and $L(\mathcal{P}) \subseteq \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}})$ is obvious. We only have to prove the following statement: if $\mathcal{P}$ is realizable, then $\mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}) \subseteq L(\mathcal{P})$.

Now suppose $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$, $n = |P|$, and $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$. Since $\mathcal{P}$ is realizable, let $\mathcal{S}' = \langle (P, M), \mathcal{A}'_1, \ldots, \mathcal{A}'_n \rangle$ be a Büchi composition which realizes $\mathcal{P}$. As $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}')$, by Theorem 2.26, we have

$$L(\mathcal{A}) = \text{JOIN}(\text{SC}_1(L(\mathcal{A}'_1)), \ldots, \text{SC}_n(L(\mathcal{A}'_n))).$$

Then by Equation 2.2, for each $i \in [1..n]$, $\pi_i(L(\mathcal{P})) \subseteq \text{SC}_i(L(\mathcal{A}'_i))$. Since $\mathcal{A}_i = \pi_i(\mathcal{P})$, the following is true:

$$\forall 1 \leq i \leq n : L(\mathcal{A}_i) \subseteq \text{SC}_i(L(\mathcal{A}'_i)) \tag{3.1}$$

Now apply SC_i on both sides of Equation 3.1. Because SC_i is idempotent, we have $\text{SC}_i(L(\mathcal{A}_i)) \subseteq \text{SC}_i(L(\mathcal{A}'_i))$ for each $i \in [1..n]$. This immediately leads to the following:

$$\text{JOIN}(\text{SC}_1(L(\mathcal{A}_1)), \ldots, \text{SC}_n(L(\mathcal{A}_n))) \subseteq \text{JOIN}(\text{SC}_1(\mathcal{L}(\mathcal{A}'_1)), \ldots, \text{SC}_n(\mathcal{L}(\mathcal{A}'_n))).$$

Hence we get $\mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}})) \subseteq \mathcal{C}(\mathcal{S}') = \mathcal{L}(\mathcal{P})$, which concludes the proof. ∎

Theorem 3.1 is a powerful tool to prove a conversation protocol is not realizable. Note that, however, the general problem of deciding whether $\mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}) = \mathcal{L}(\mathcal{A})$ may not be decidable.

3.1.2 Realizability Analysis

This section presents the main result of this chapter. We introduce three sufficient conditions that can guarantee the realizability of a conversation protocol. We write $w_1 \preceq w_2$ to denote a word w_1 being a prefix of w_2 (w_1 may be equal to w_2). Let $L^*_{\preceq}(\mathcal{A})$ include all finite prefixes of $L(\mathcal{A})$ for a Büchi automaton $\mathcal{A}$. Obviously, $L^*_{\preceq}(\mathcal{A})$ is regular. For each nonredundant Büchi automaton $\mathcal{A}$ we can construct a standard FSA, written as $\mathcal{A}^*$, to recognize $L^*(\mathcal{A})$. $\mathcal{A}^*$ can be constructed by making each state of $\mathcal{A}$ a final state.

Lossless join condition: Let us first study the following motivating example for the lossless join condition.

Example 3.2 Consider a composition schema $S = (P, M)$ with four peer prototypes, p_1, p_2, p_3, p_4, where $M_1^{out} = M_2^{in} = \{\alpha\}$, $M_3^{out} = M_4^{in} = \{\beta\}$, and $M_1^{in} = M_2^{out} = M_3^{in} = M_4^{out} = \emptyset$. Let $\mathcal{P} = \langle S, \mathcal{A}\rangle$ be a Büchi conversation protocol where $\mathcal{A} = (M, T, s, F, \Delta)$ is a Büchi automaton with $M = \{\alpha, \beta\}$, $T = \{0, 1, 2\}$, $s = 0$, $F = \{2\}$, and $\Delta = \{(0, \alpha, 1), (1, \beta, 2), (2, \beta, 2)\}$. $\mathcal{P}$ is not realizable, because there is no communication between p_1 and p_3, hence, there is no way for them to make sure that α is sent before any β is sent[1]. ∎

A formal proof of why $\mathcal{P}$ is not realizable can be based on Theorem 3.1. Consider the projected composition $\mathcal{S}^{\text{PROJ}}_{\mathcal{P}}$ of the conversation protocol $\mathcal{P}$ in Example 3.2. $\mathcal{S}^{\text{PROJ}}_{\mathcal{P}}$ can generate a conversation $\beta\alpha\beta^\omega$ which does not belong to $L(\mathcal{P})$. By Theorem 3.1, $\mathcal{P}$ is not realizable. Motivated by the fact that $\beta\alpha\beta^\omega$ belongs to $\text{JOINC}(L(\mathcal{P}))$, we propose a *lossless join* condition to enforce a

[1]Notice that Example 3.2 is essentially the reactive version of Example 2.19.

conversation protocol to be "complete" so that it includes all words in the join of its projections to peer prototypes. For example, when an additional transition $(0, \beta, 0)$ is added into the transition relation of $\mathcal{P}$ in Example 3.2, it becomes complete (w.r.t. the join of projections) and realizable.

Definition 3.2 *A Büchi conversation protocol $\mathcal{P}$ is* lossless join *if*

$$L(\mathcal{P}) = \text{JOINC}(L(\mathcal{P})).$$

Given a Büchi conversation protocol $\mathcal{P}$, the decision procedure for the lossless join condition is straightforward. Obtain $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ from $\mathcal{P}$, and then construct the Cartesian product (a generalized Büchi automaton [45] with multiple sets of accepting states) of the projections $\mathcal{A}_1, \ldots, \mathcal{A}_n$ that are projected from $\mathcal{P}$. Then verify whether the resulting product is equivalent to $\mathcal{A}$. Since the resulting product may contain ϵ transitions, we have to first check if it accepts words of finite length (note that the original protocol accepts infinite words only) and then eliminate ϵ-transitions from the product. The check of acceptance of finite words is to look for states from which there is an infinite ϵ path traveling through at least one final state infinitely many times. The elimination of ϵ-transitions can be achieved by ϵ-transition elimination algorithm for standard FSA. After eliminating ϵ-transitions, we can convert the product from a generalized Büchi automaton to a standard Büchi automaton, and then conduct the equivalence check.

Synchronous compatible condition: Let us first revisit the Fresh Market Update example presented in Chapter 2. For the convenience of reading, Fig. 3.1 presents the peer implementations of the Fresh Market Update example, which

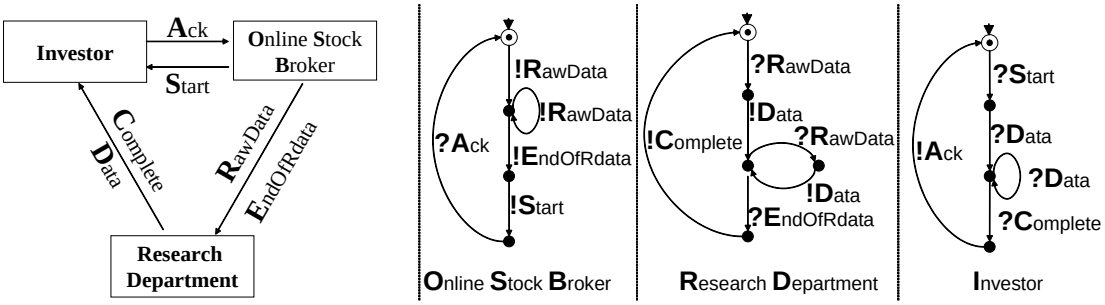


Figure 3.1. Fresh Market Update Service

originally appeared in Fig. 2.8. We have argued in Chapter 2 that the example is bad because its conversation set is not ω-regular. In fact it is even worse. Consider the peer Investor, it is possible that message **Data** arrives earlier than **Start**, and then Investor gets stuck. This scenario (under the asynchronous communication assumption) is similar to the case of "illegal state" [3] in the *synchronous composition* of peers. During the synchronous composition of a set of peers, for each message transmitted, its sender and receiver must synchronize their send and receive actions, and hence peers do not need input queues to store incoming messages because they are consumed immediately. The synchronous composition in [3] is actually the Cartesian product of peers in our context. Formally, given a Cartesian product of peers, an illegal state, is a state in the product where some peer is ready to send out a message α but the receiver of α is not ready to receive it (i.e., none of the transitions starting from the local state of the receiver consumes the message α). In the following, we define the *synchronous compatible* condition, to prevent illegal states in the synchronous composition of peers. We will show later, when combined with other conditions, the synchronous compatible condition ensures realizability for conversation protocols, and in addition, it prevents peers getting stuck for non-reactive web service compositions.

Given a Büchi composition $\mathcal{S} = \langle (M, P), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, $\mathcal{S}$ is said to be *synchronous compatible* if the Cartesian product of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ does not have illegal state. For example, we can make the Fresh Market Update example synchronous compatible by the following changes: (1) introduce an additional message **A**ck2 to synchronize the three peers in a lock-step fashion (i.e., for each **D**ata received, Investor sends an **A**ck2 to Online Stock Broker, and Online Stock Broker does not send out the next **R**awData until the **A**ck2 is received), and (2) move the transition for **S**tart in Online Stock Broker so that **S**tart is sent before the first **R**awData in each round of message exchange.

A conversation protocol $\mathcal{P}$ is synchronous compatible if its "determinized"[2] $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ is synchronous compatible. Formally, the condition is defined as below.

Definition 3.3 *Let $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ be a Büchi conversation protocol, and $n = |P|$. $\mathcal{P}$ is said to be* synchronous compatible *if for each word $w \in M^*$ and each message $\alpha \in M_a^{out} \cap M_b^{in}$ for $a, b \in [1..n]$, the following holds:*

$$(\forall i \in [1..n],\ \pi_i(w) \in \pi_i(L_{\preceq}^*(\mathcal{A}))) \ \wedge\ \pi_a(w\alpha) \in \pi_a(L_{\preceq}^*(\mathcal{A}))$$
$$\Rightarrow \quad \pi_b(w\alpha) \in \pi_b(L_{\preceq}^*(\mathcal{A})).$$

The decision procedure of the synchronous compatible condition for a Büchi conversation protocol $\mathcal{P}$ proceeds as follows: construct $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ from $\mathcal{P}$. Treat every peer in $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ as a standard FSA, and make each state a final state, and then determinize each peer. Construct the Cartesian product of all peers, and check if there is any illegal state. $\mathcal{P}$ is not synchronous compatible if an illegal state is found.

[2]Note that a nondeterministic Büchi automaton can not always be determinized. In the check of synchronous compatible condition, each Büchi peer is regarded as a standard FSA when it is determinized.

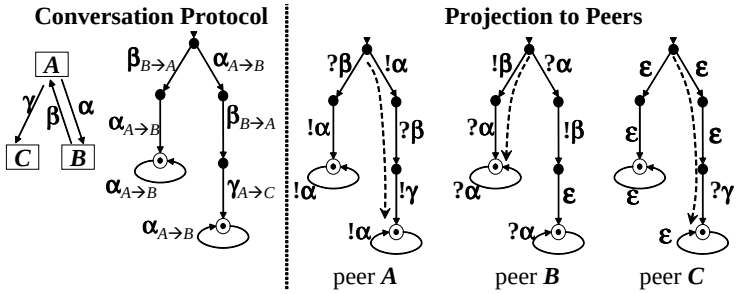


Figure 3.2. Ambiguous Execution

Autonomous condition: The combination of lossless join and synchronous compatibility is still not strong enough to guarantee a realizable protocol. Consider the example presented in Fig. 3.2. On the left side of the figure is the Büchi conversation protocol, and on the right side is its projection to each peer. It is easy to verify that the protocol in Fig. 3.2 satisfies both lossless join and synchronous compatible condition. However it is not a realizable protocol. Think about one possible execution of the composition of these peers. At the beginning, peer B sends a message β to peer A, and β is stored in the input queue of peer A. Then peer A sends message α to peer B, and $\beta\alpha$ is recorded by the global watcher. Now peer B continues to execute the left path of the protocol, consumes the α in the queue; and peer A executes the right path of the protocol, consumes the β, and sends out γ. Eventually, a non-specified conversation $\beta\alpha\gamma\alpha^{\omega}$ is generated, without being noticed by any of the peers involved. By Theorem 3.1, it directly follows that the conversation protocol on the left of Fig. 3.2 is not realizable.

Take a close look at the execution paths of all peers, which are shown using dotted arrows in Fig. 3.2. It is clear that the abnormal conversation is the

result of "ambiguous" understanding of the protocol by peers, and the racing between A and B at the initial state is the main cause. Consequently, we introduce an *autonomous* condition to restrict racing conditions, so that at any point each peer has exactly one choice to receive, or to send, or to terminate (unlike in Fig. 3.2 peer A can either send out α or receive β at the initial state). Note that here the "determinism" about send/receive/terminate action does not restrict the nondeterministic decision on which message to send, once the next action is determined to be a send action. In the following we formally define this autonomous condition.

Let $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ be a Büchi conversation protocol and $n = |P|$. A peer prototype $p_i \in P$ is *output-ready* (*input-ready*) at a word $w \in M_i^*$ if there exists a word $w'\alpha \in L^*_{\preceq}(\mathcal{A})$ such that α is an output (respectively, input) message of p_i and $\pi_i(w') = w$. Similarly p_i is *terminate-ready* at a word $w \in M_i^*$ if there exists a word $w' \in L(\mathcal{A})$ such that $\pi_i(w') = w$.

Definition 3.4 *A Büchi conversation protocol $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ is* autonomous *if for each peer prototype $p_i \in P$ and for each finite prefix $w \in L^*_{\preceq}(\mathcal{A})$, p_i at $\pi_i(w)$ is exactly one of the following: output-ready, input-ready, or terminate-ready.*

we can check the autonomous condition for a Büchi conversation protocol $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ as follows. For each peer prototype $p_i \in P$, let $\mathcal{A}_i = (M_i, T_i, s_i, F_i, \Delta_i)$ be the corresponding peer in $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$, and let $T_i' \subseteq T_i$ include each state s where an infinite ϵ path starting at s passes at least one final state for infinitely many times. Construct prefix automaton $\mathcal{A}_i^*$ for each $\mathcal{A}_i$ by making each state in $\mathcal{A}_i$ a final state. Determinize $\mathcal{A}_i^*$ by a standard determinization algorithm for finite state automata. Each state s' of determinized $\mathcal{A}_i^*$ corresponds to a subset of T_i,

and we denote it by $T_i(s')$. For each state s', when $T_i(s') \cap T_i'$ is not empty, we require that there is no outgoing transitions starting from s'. If $T_i(s') \cap T_i'$ is empty, then the outgoing transitions from s' are required to be either all for output messages or all for input messages. The complexity of the above check is EXPTIME because of the determinization procedure. The following lemma summarizes the complexity of checking three realizability conditions.

Lemma 3.3 To check if a Büchi conversation protocol $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$ satisfies the lossless join, synchronous compatible, and autonomous conditions can be determined in EXPTIME in the size of $\mathcal{A}$.

We now proceed to present the main result (Theorem 3.5), which shows that if the realizability conditions are satisfied, a conversation protocol is realizable.

Lemma 3.4 If a Büchi conversation protocol $\mathcal{P}$ is synchronous compatible and autonomous, the following two statements are both true for each conversation $w \in \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}})$.

1. during any complete run of w, each message is consumed eagerly, i.e., a peer never sends or terminates when its queue is not empty, and
2. for each peer prototype p_i, $\pi_i(w) \in \pi_i(L(\mathcal{P}))$.

Proof: We first introduce an important fact about "eager consumption" of messages. Given a run γ of a Büchi composition $\langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, for each peer $\mathcal{A}_i$, the *word generated by (the local run of)* $\mathcal{A}_i$ *during* γ is the path traversed by $\mathcal{A}_i$ during γ. For example, consider the run which generates the word $\beta\alpha\gamma\alpha^\omega$ in Fig. 3.2. The local run of peer A generates $\alpha\beta\gamma\alpha^\omega$, which is the path pointed

by the dotted arrows in Fig. 3.2. Since a word generated by the local run of a peer $\mathcal{A}_i$ is the path traversed by $\mathcal{A}_i$, it is not hard to see that the word always belongs to $L(\mathcal{A}'_i) \cup L^*_{\preceq}(\mathcal{A}_i)$ (where $\mathcal{A}'_i$ is the Büchi automaton generated from $\mathcal{A}_i$ by making every state a final state). If each message is consumed eagerly during γ, it is not hard to see that for the conversation of γ, its projection to each peer is the word generated by that peer during γ. (Note that if a message is not consumed "eagerly", the above property is not guaranteed. For example, as we have argued earlier, for the conversation $\beta\alpha\gamma\alpha^\omega$ generated by Fig. 3.2, the local run of peer A is $\alpha\beta\gamma\alpha^\omega$ which is not the projection of the conversation to A. This is because message β is not consumed eagerly and it is stored in the queue of A when α is being sent.) From the above fact, it is not hard to see that if statement (1) is true, then statement (2) is true.

Now we concentrate on the proof for statement (1) by contradiction. Assume that for conversation $w = \alpha_0\alpha_1\ldots$ there is a complete run γ where a message α_m from $\mathcal{A}_x$ to $\mathcal{A}_y$ is the first message that is not consumed eagerly. Since for each $a < m$, α_a is consumed eagerly by its receiver, the projection of word $\alpha_0 \ldots \alpha_{m-1}$ to each peer is the word generated by that peer during γ. Hence for each peer $\mathcal{A}_i$ we have $\pi_i(\alpha_0 \ldots \alpha_{m-1}) \in \pi_i(L(\mathcal{A}^*))$. We also know that since the input queue of $\mathcal{A}_x$ is empty when it sends out α_m, $\pi_x(\alpha_0 \ldots \alpha_m)$ is contained in $\pi_x(L(\mathcal{A}^*))$. Now, by synchronous compatible condition, $\pi_y(\alpha_0 \ldots \alpha_m)$ should also be contained in $\pi_y(L(\mathcal{A}^*))$, hence $\mathcal{A}_y$ is input ready at word $\pi_y(\alpha_0 \ldots \alpha_{m-1})$.

According to the assumption that message α_m is not consumed eagerly, there are three possibilities: 1) peer $\mathcal{A}_y$ sends out a message (let it be α_n) during run γ when α_m is still in its input queue, 2) peer $\mathcal{A}_y$ terminates and never consumes α_m, and 3) peer $\mathcal{A}_y$ is stuck by message α_m. As Cases 2 and 3 are directly

refuted by their conflicts with the definition of a complete run, in the following, we discuss Case 1 only. Now, consider peer $\mathcal{A}_y$ in the run γ. Its local run up to the send of α_n generates the word $\pi_y(\alpha_0 \dots \alpha_{m-1}\alpha_n)$ and hence $\pi_y(\alpha_0 \dots \alpha_{m-1}\alpha_n) \in \pi_y(L(\mathcal{A}^*))$. Thus peer $\mathcal{A}_y$ is output ready at word $\pi_y(\alpha_0 \dots \alpha_{m-1})$. Combined with the proved fact that $\mathcal{A}_y$ is input ready at the same word, this contradicts with the autonomous condition. Therefore the assumption is false, and statement (1) is true. ∎

The statement (2) of Lemma 3.4 implies the following theorem.

Theorem 3.5 A Büchi conversation protocol is realizable if it satisfies the lossless join, synchronous compatible, and autonomous conditions.

Following Lemma 3.3 and Theorem 3.5, we get a three step specification and verification strategy. (1) A conversation protocol is specified by a realizable Büchi automaton; (2) Desired properties are verified on the conversation protocol; (3) The peer implementations are synthesized from the conversation protocol.

3.1.3 Discussion of Realizability Conditions

In the following we discuss several interesting issues about the realizability conditions presented in this chapter. First we show that all the three realizability conditions are independent concepts, i.e., neither of them can be expressed as a boolean combination of the other two conditions. Next, we prove that the three conditions are not redundant, in the sense that, for any two of the three realizability conditions, there exists a non-realizable Büchi conversation protocol which satisfies those two realizability conditions but not the remaining realiz-

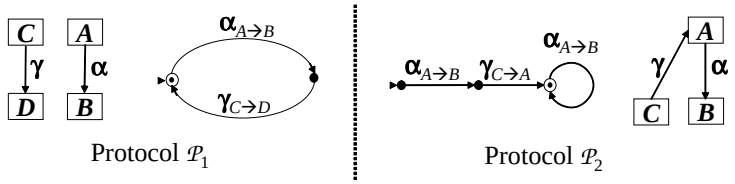


Figure 3.3. Examples for Lossless Join and Synchronous Compatibility

ability condition. In other words, missing any one of the realizability conditions will not guarantee realizability. Finally, we show how these conditions fit into practical applications.

In Fig. 3.3, we show two non-realizable Büchi conversation protocols $\mathcal{P}_1$ and $\mathcal{P}_2$. It is easy to verify that $\mathcal{P}_1$ satisfies autonomous condition and synchronous compatible condition, however it violates the lossless join condition, because word $(\gamma\alpha)^\omega$ is contained in $\text{JOIN}(\pi_A(L(\mathcal{P}_1)), \ldots, \pi_D(L(\mathcal{P}_1)))$. Also note that $(\gamma\alpha)^\omega$ will be generated by $\mathcal{S}^{\text{PROJ}}_{\mathcal{P}_1}$, and by Theorem 3.1, $\mathcal{P}_1$ is not realizable.

$\mathcal{P}_2$ fails the synchronous compatible condition. The reason is that for empty word ϵ, $\pi_C(\epsilon\gamma)$ is contained in $\pi_C(L(\mathcal{P}_2^*))$ however $\pi_A(\epsilon\gamma) \notin \pi_A(L(\mathcal{P}_2^*))$. Concerning the lossless join condition and autonomous condition, it is not hard to see that $\mathcal{P}_2$ satisfies both of them. Consider $\mathcal{S}^{\text{PROJ}}_{\mathcal{P}_2}$, it is not hard to see that $\gamma\alpha\alpha^\omega$ is one of the conversation generated by $\mathcal{S}^{\text{PROJ}}_{\mathcal{P}_2}$, however it not contained in $L(\mathcal{P}_2)$, and hence by Theorem 3.1, $\mathcal{P}_2$ is not realizable.

As we have shown, the non-realizable conversation protocol in Fig. 3.2 is one example where synchronous compatible and lossless join conditions are satisfied while autonomous condition is violated. As a summary of the above discussion of Fig. 3.2 and Fig. 3.3, we have the following two propositions.

Proposition 3.6 Each of the lossless join, synchronous compatible, and au-

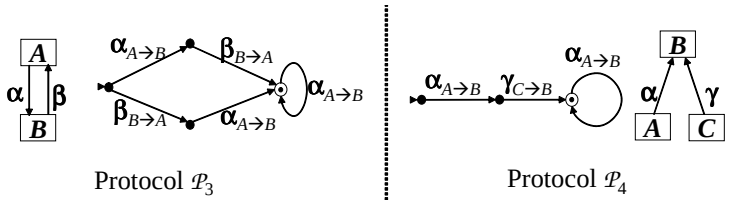


Figure 3.4. Two More Examples

tonomous conditions is not equivalent to any boolean combination of the other two conditions.

Proposition 3.7 For each of the lossless join, synchronous compatible and autonomous conditions, there exists a non-realizable conversation protocol which violates that condition while satisfying the other two.

Now, the next question is, how restrictive are the realizability conditions? We present some facts as a partial answer to this question. It is not hard to see that lossless join condition is a necessary condition for the realizability; however autonomous condition is not. For example, the protocol $\mathcal{P}_3$ in Fig. 3.4 is realizable but not autonomous. Synchronous compatible condition is not a necessary condition for realizability either. Consider the protocol $\mathcal{P}_4$ shown in Fig. 3.4. It is not synchronous compatible because at the initial state peer B is not receptive to message γ. However, the protocol is realizable, because $\alpha\gamma\alpha^\omega$ is the only conversation successfully generated by the composition of peers which follow the protocol. (Note that, during the run of $\gamma\alpha^\omega$, peer B is always stuck and hence the word does not count as a conversation.) The Proposition 3.8, given below, summarizes the above discussion.

Proposition 3.8 Lossless join is a necessary condition for realizability while autonomous condition and synchronous compatible condition are not.

The three realizability conditions in Theorem 3.5 may seem restrictive, however, they are satisfied by many real life web service applications. We verified that four out of the six examples listed on the IBM Conversation Support site [51] satisfy the conditions, and the other two examples both violate the autonomous condition. For instance, one "Meta Conversation" example in [51], allows the two peers in a meta conversation to race at the beginning to decide who initiates the conversation first. Unfortunately, our autonomous condition forbids such racing. In fact, except restricting the racing between send and receive actions, our realizability conditions allow a certain level of parallelism, which makes it acceptable for many web service applications.

3.2 Realizability Analysis for FSA Conversation Protocols

For FSA conversation protocols, the notions of "lossless join", "synchronous compatible", and "autonomous" conditions can be defined by replacing the "Büchi" with "FSA", in Definition 3.2, 3.3, 3.4, respectively. For FSA conversation protocols, the decision procedures for the synchronous compatible and autonomous conditions are the same as those for Büchi conversation protocols. In addition, the check for the lossless join condition is even simpler, because we do not have to check if the Cartesian product accepts both infinite and finite words, and we do not have to convert from a generalized Büchi automaton to

a standard Büchi automaton (because the Cartesian product of FSA is a single standard FSA).

The non-reactive version of Lemma 3.4 and Theorem 3.5 are as follows. We omit the proof for these two results here, because they are exactly the same as the proof for their reactive version (notice that the change of acceptance condition of automata does not affect each proof).

Lemma 3.9 If an FSA conversation protocol $\mathcal{P}$ is synchronous compatible and autonomous, the following two statements are both true for each conversation $w \in \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}})$.

1. during any complete run of w, each message is consumed eagerly, i.e., a peer never sends or terminates when its queue is not empty, and
2. for each peer prototype p_i, $\pi_i(w) \in \pi_i(L(\mathcal{P}))$.

Theorem 3.10 An FSA conversation protocol is realizable if it satisfies the lossless join, synchronous compatible, and autonomous conditions.

Note that the result of Theorem 3.10 is still not good enough. Although an FSA conversation protocol can be realized by its projections, there is still no guarantee that the composition of projections does not generate "bad runs" where a peer can be stuck by an unspecified message event. In the following we present a strengthened result for FSA conversation protocols. We show that when realizability conditions are satisfied, we determinize each projection of the conversation protocol, and every partial run of the composition of these deterministic projections is a completable run. Interestingly, the same result cannot be applied to

reactive conversation protocols, due to the inequivalence of nondeterministic and deterministic Büchi automata.

Theorem 3.11 Given a synchronous compatible and autonomous FSA conversation protocol $\mathcal{P} = \langle (P, M), \mathcal{A} \rangle$, let Let $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ}} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be the *determinized projected composition* which is obtained by determinizing each peer in $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ}}$, then the following statements are true:

1. During any (complete or partial) run of $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ}}$, for each configuration, if the input queue of a peer $\mathcal{A}_i$ is not empty and let α be the message at the head of the queue, $\mathcal{A}_i$ must be in a non-final state which is receptive to α.
2. If both $\mathcal{A}$ and $\mathcal{A}^*$ are lossless join, then every run of $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ}}$ is completable.

Proof: We assume that γ is the shortest run violating statement (1), and let $w = \alpha_0 \alpha_1 \ldots \alpha_m$ be the corresponding watcher content. Let $\mathcal{A}_x$, $\mathcal{A}_y$ be the sender and receiver of α_m respectively. By a similar argument of Lemma 3.4, we can show that $\mathcal{A}_y$ is input ready (for message α_m) at $\pi_y(\alpha_0 \ldots \alpha_{m-1})$, and $\pi_y(\alpha_0 \ldots \alpha_{m-1})$ is the word generated by the local run of $\mathcal{A}_y$. As $\mathcal{A}_y$ is a Deterministic Finite State Automata (DFA), after running $\pi_y(\alpha_0 \ldots \alpha_{m-1})$, it advances to a unique state in DFA, and this state is receptive to message α_m. In addition, by autonomous condition the state is not a final state. Hence the assumption is false and we have proved statement (1).

We provide a constructive proof for statement (2). Given a (partial) run γ of $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ}}$, and w the corresponding watcher content, we can construct a complete run where γ is its prefix. According to statement (1), each message ever sent is receptive by its receiver. Thus, by making each peer consume its queue content, γ

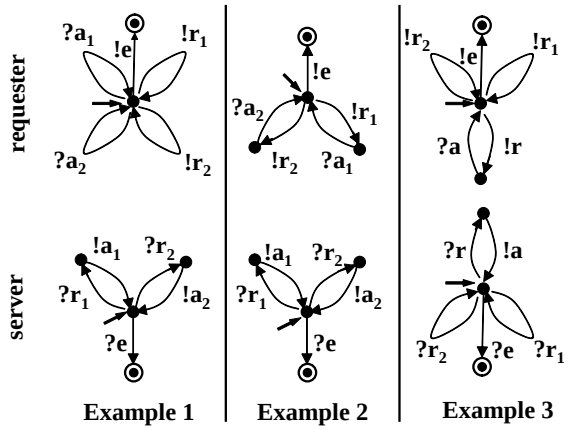


Figure 3.5. Three Motivating Examples

can be extended to a run γ' so that all queues are empty in the last configuration of γ'. Now for each peer $\mathcal{A}_a$, $\pi_a(w) \in \pi_a(L(\mathcal{A}^*))$. Since $\mathcal{A}^*$ is lossless join, w is contained in $L(\mathcal{A}^*)$, i.e, w is a prefix of some word $w' \in L(\mathcal{A})$. We can always find a run η for w' s.t. during η each message is consumed immediately after it is sent. Concatenate γ' and the part of η after producing w in the watcher, we get an extension of γ which is a complete run. ∎

Theorem 3.11 suggests that specifying peers using DFSA has special benefits in avoiding unspecified message receptions. In addition, making the prefix automaton lossless join can avoid dead-lock. These are both very natural and non-restrictive requirements for web service designs.

3.3 Synchronizability Analysis

This section presents a synchronizability analysis for bottom-up specified web service compositions. Synchronizability analysis can help us avoid undecidability that is caused by the asynchronous communication. We start with a motivating

example, and then we introduce the technical results of the analysis. Finally, we present a relaxation of autonomous condition, which can also be applied to realizability analysis.

Consider the three example FSA compositions given in Fig. 3.5. Each FSA composition consists of two peers: a requester and a server. For each "request" message (represented as r_i) sent by the requester, the server will respond with a corresponding "acknowledgment" (a_i). However this response may not be immediate (e.g. in Example 1). Finally the "end" message (e) concludes the interaction between the requester and the server.

In [40], we can verify properties of these examples by translating them to Promela. However, as discussed in [40], we need to bound the sizes of the input queues (communication channels in Promela) to be able to verify a web service composition using SPIN, since it is a finite state model checker. In fact, based on the undecidability of LTL verification (Chapter 2), it is generally impossible to verify the behavior of a web service composition with unbounded queues. In general, best we can do is *partial* verification, i.e., to verify behavior of a web service composition for queues with a fixed length. Note that the absence of errors using such an approach does not guarantee that the web service composition is correct. Interestingly, in this section we will show that, Examples 2 and 3 are different from Example 1 in Fig. 3.5 in that the properties of Examples 2 and 3 can in fact be verified for unbounded message queues, whereas for Example 1 we can only achieve partial verification.

First, note that in Example 1 the requester can send an arbitrary number of messages before the server starts consuming them. Hence the conversation set

of Example 1 is not a regular set [16]. Actually it is a subset of $(r_1|r_2|a_1|a_2)^*e$ where the number of r_i and a_i messages are equal and in any prefix the number of r_i messages is greater than or equal to the number of a_i messages [16]. It is not surprising that we cannot map the behavior of Example 1 to a finite state process. Another problem with Example 1 is the fact that its state space increases exponentially with the sizes of the input queues. Hence, even partial verification for large queue sizes becomes intractable.

In Example 2 the requester and server processes move in a lock-step fashion, and it is easy to see that the conversations generated by Example 2 is $(r_1a_1 \mid r_2a_2)^*e$, i.e., a regular set. In fact, the web service composition described in Example 2 has a finite set of reachable states. During any execution of Example 2, at any state, there is at most one message in each queue. Based on the results we will present in this section, we can statically conclude that properties of Example 2 can be verified using synchronous communication (in other words, using input queues of size 0).

Unlike Example 2, Example 3 has an infinite state space as Example 1. In other words, the number of messages in the input queues for Example 3 is not bounded. Similar to Example 1, the state space of Example 3 also increases exponentially with the sizes of the queues. However, unlike Example 1, the conversation set of Example 3 is regular. Although Example 3 has an infinite state space, we will show that the properties of Example 3 can also be verified for arbitrary queue sizes.

We can experimentally demonstrate how state spaces of the examples in Fig. 3.5 change with the increasing queue sizes. In Fig. 3.6 we present the size of

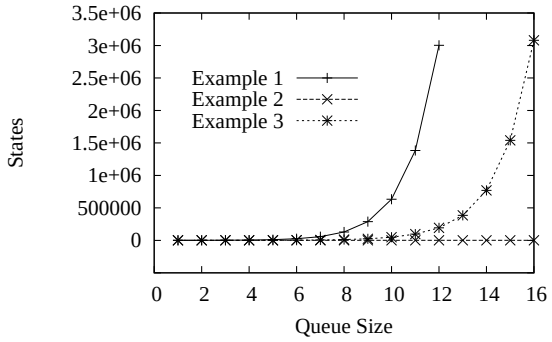


Figure 3.6. State Space and Queue Size

the reachable state space for the examples in Fig. 3.5 computed using the SPIN model checker for different input queue sizes. The x-axis of the figure is the size of the input queues, and y-axis displays the number of reachable states computed by SPIN. As shown in the figure, the state space of Example 2 is fixed (always 43 states), however the state spaces of Examples 1 and 3 increase exponentially with the queue size. Below we will show that we can verify behaviors of Examples 2 and 3 for arbitrary queue sizes, although best we can do for Example 1 is partial verification. In particular, we will show that the communication among peers for Examples 2 and 3 are "synchronizable" and we can verify their properties using synchronous communication and guarantee that the verified properties hold for asynchronous communication with unbounded queues.

3.3.1 Synchronous Communication

To further explore the differences of Examples 2 and 3 from Example 1, we define an alternative "synchronous" semantics for web service compositions different than the one in Section 2. Intuitively, the synchronous semantics restricts that each peer consumes its incoming messages immediately. Therefore, there is

no need to have the input message queue.

Recall that an FSA composition $\mathcal{S}$ is a tuple $\mathcal{S} = \langle (P, M), \mathcal{A}_1, ..., \mathcal{A}_n \rangle$ where each automaton $\mathcal{A}_i$ describes the behavior of a peer. In a global configuration $(Q_1, t_1, ..., Q_n, t_n, w)$ of $\mathcal{S}$, Q_j's ($j \in [1..n]$) are the configurations of the input queues. We now define a *configuration* of an FSA composition *with the synchronous communication semantics*, or *sc-configuration*, as a tuple $(t_1, ..., t_n, w)$, which differs from a configuration by dropping all input queues.

When peers interact with each other through asynchronous communication, a send operation inserts a message to the input queue of the target peer and a receive operation removes the message at the head of the input queue. The definition of the *derivation* relation between two sc-configurations is modified from the asynchronous case so that a send transition can only be executed instantaneously with a matching receive operation, i.e., sending and receiving of a message occur synchronously. We call this semantics the *synchronous communication semantics* for an FSA web service composition.

The definitions of the watcher and the conversation set are modified accordingly. In particular, given an FSA composition $\mathcal{S}$, let $\mathcal{C}_{\text{syn}}(\mathcal{S})$ denote the conversation set under the synchronous communication semantics. An FSA composition is *synchronizable* if its conversation set remains the same when the synchronous communication semantics is used, i.e., $\mathcal{C}(\mathcal{S}) = \mathcal{C}_{\text{syn}}(\mathcal{S})$.

Clearly, if an FSA composition is synchronizable, then we can verify its behavior without any input queues and the results of the verification will hold for the behaviors of the FSA composition in the presence of asynchronous communication with unbounded queues. In the following we will give sufficient conditions

for synchronizability. Based on these conditions, we can show that Examples 2 and 3 in Fig. 3.5 are indeed synchronizable.

3.3.2 Synchronizability Analysis

We propose two sufficient synchronizability conditions to identify synchronizable web service compositions. These two conditions are essentially the two conditions in the realizability analysis for conversation protocols.

1) Synchronous compatible condition: An FSA composition $\langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ is synchronous compatible if for each $i \in [1..n]$, each word $w \in M^*$, and each message $\alpha \in M_a^{out} \cap M_b^{in}$:

$$(\forall i \in [1..n]\ \pi_i(w) \in L(\mathcal{A}_i^*)) \ \wedge\ \pi_a(w\alpha) \in L(\mathcal{A}_a^*) \ \Rightarrow\ \pi_b(w\alpha) \in L(\mathcal{A}_b^*),$$

where $\mathcal{A}_i^*$ is the prefix automaton of $\mathcal{A}_i$.

2) Autonomous condition: An FSA composition $\langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ is autonomous if for each peer $\mathcal{A}_i$, and for each word $w \in M_i^*$, exactly one of the following three statements holds: (a) w is accepted by $\mathcal{A}_i$. (b) there exists $\beta \in M_i^{in}$ s.t. $w\beta \in L(\mathcal{A}_i^*)$. (c) there exists $\alpha \in M_i^{out}$ s.t. $w\beta \in L(\mathcal{A}_i^*)$.

Theorem 3.12 *An FSA web service composition is synchronizable if it satisfies the synchronous compatible and autonomous conditions.*

Proof: Let $\mathcal{S}$ be an FSA composition which satisfies the synchronous compatible condition and the autonomous condition. By Lemma 3.9, the projection of each conversation to a peer prototype p_i is an accepted word of $\mathcal{A}_i$. This immediately leads to the following:

$$\mathcal{C}(\mathcal{S}) \subseteq \textsc{join}(L(\mathcal{A}_1), \ldots, L(\mathcal{A}_n)) \tag{3.2}$$

Now by Theorem 2.16, the conversation set of $\mathcal{S}$ can be captured by the formula:

$$\mathcal{C}(\mathcal{S}) = \text{JOIN}(\text{LP}_1^*(L(\mathcal{A}_1)), \ldots, \text{LP}_n^*(L(\mathcal{A}_n))) \tag{3.3}$$

Since for each $i \in [1..n]$, $L(\mathcal{A}_i) \subseteq \text{LP}_i^*(L(\mathcal{A}_i))$, combining Equations 3.2 and 3.3, we have $\mathcal{C}(\mathcal{S}) = \text{JOIN}(L(\mathcal{A}_1), \ldots, L(\mathcal{A}_n))$, which is accepted by the Cartesian product (i.e., the synchronous composition) of all peers. ∎

Note that both Examples 2 and 3 in Fig. 3.5 are synchronizable whereas Example 1 is not (it violates the autonomous condition). Hence, we can verify the properties of Examples 2 and 3 using synchronous communication (which can be achieved in SPIN by restricting the communication channel lengths to 0) and the results we obtain will hold for behaviors generated using asynchronous communication with unbounded queues.

Similar to the realizability on FSA conversation protocols, synchronizability analysis on FSA compositions also has additional good properties. The results are summarized in the following theorem.

Theorem 3.13 Given a synchronous compatible and autonomous FSA composition $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, if for each $i \in [1..n]$, $\mathcal{A}_i$ is a Deterministic Finite State Automaton, then the following statements are true:

1. During any (complete or partial) run of $\mathcal{S}$, for each configuration, if the input queue of a peer $\mathcal{A}_i$ is not empty and let α be the message at the head of the queue, $\mathcal{A}_i$ must be in a non-final state which is receptive to α.

2. Let $\mathcal{A}$ be the (non-minimized) Cartesian product of $\mathcal{A}_1, \ldots, \mathcal{A}_n$. If $\mathcal{A}$ has no deadlock, then $\mathcal{S}$ is free of deadlock.

Proof: The proof for statement (1) is exactly the same as the proof for the statement (1) of Theorem 3.11. Now we prove the statement (2) by contradiction. If there is a deadlock in a run γ for $\mathcal{S}$, by the definition of deadlock, at the last configuration of γ, each peer stays in a state where either there is no outgoing transition, or all outgoing transitions are receive-actions. Since each peer is a DFSA, by statement (1), during the run γ, each message is consumed *eagerly*, hence for γ we can always construct a γ' where each message is consumed *immediately* (i.e., the size of the input queue of each peer never exceeds 1). By merging the neighboring (and also the corresponding) pair of send and receive actions in γ', we get a run in the Cartesian product which leads to a sc-configuration (where each peer stays in the same local state as that of the last configuration of γ). Now this sc-configuration is a deadlock state, which contradicts with the assumption that the Cartesian product is free of deadlock. ∎

Relaxing the Autonomy Condition. During our effort to translate from BPEL web services to automata model in [40], the `flow` construct in BPEL specification generates the Cartesian product of its flow branches when it is translated to the automata. Unfortunately, such `flow` constructs are likely to violate the autonomous condition given above. For example, assume that there are two branches inside a flow statement, and each branch is a single `invoke` operation which first sends a request and then receives response. In the automaton translation, there will be a state with one transition for sending out the request for one of the branches and another transition for receiving the response for the other branch. Note that such a state violates the autonomous condition. However, even the corresponding peer sends out a message while its input queue is not empty,

since the Cartesian product of the flow branches includes all the permutations of the transitions in different branches we can show that there is an equivalent computation where the send operation is executed when the queue is empty, after the receive operation. We can generalize this scenario and we can relax the autonomous condition to single-entry single-exit permutation blocks. A permutation block has no cycles and no final states and contains all the permutations of the transitions from its entry to its exit. Then, we relax the autonomous condition by stating that all the states in a permutation block (including the entry but excluding the exit) satisfy the autonomous condition.

3.3.3 Synchronizability Analysis for Büchi Compositions

The synchronizability analysis for Büchi compositions has exactly the same results as FSA compositions, because the different acceptance condition for Büchi automata does not affect the proof for the reactive version of Theorem 3.12 and Theorem 3.13. The only difference is that the Cartesian product of Büchi peers is a generalized Büchi automaton, which can be converted into a standard Büchi automaton. We simply list these two theorems in the following, and omit the proof for them.

Theorem 3.14 A Büchi web service composition is synchronizable if it satisfies the synchronous compatible and autonomous conditions.

Theorem 3.15 Given a synchronous compatible and autonomous Büchi composition $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, if for each $i \in [1..n]$, $\mathcal{A}_i$ is a deterministic Büchi automaton, then the following statements are true:

1. During any (complete or partial) run of $\mathcal{S}$, for each configuration, if the input queue of a peer $\mathcal{A}_i$ is not empty and let α be the message at the head of the queue, $\mathcal{A}_i$ must be in a state which is receptive to α, and which does not have an infinite ϵ-path traversing through final states.

2. Let $\mathcal{A}$ be the (non-minimized) Cartesian product of $\mathcal{A}_1, \ldots, \mathcal{A}_n$. If $\mathcal{A}$ has no deadlock, then $\mathcal{S}$ is free of deadlock.

3.4 Related Work

To the best of our knowledge, the notion of realizability on open/concurrent systems was first studied in the late 80's (see [1, 70, 71]). In [1, 70, 71], realizability problem is defined as whether a peer has a strategy to cope with the environment no matter how the environment decides to move. The concept of realizability studied in this chapter is rather different. In our model, the environment of an individual peer consists of other peers whose behaviors are also governed by portions of the protocol relevant to them. In addition, our realizability requires that implementation should generate *all* (instead of a subset of) behaviors as specified by the protocol.

A closer notion to the realizability in this chapter is the concept of "weak realizability" of Message Sequence Chart (MSC) Graphs studied in [7]. However, the MSC Graph model captures both "send" and "receive" events, while in our web service composition model we are interested in the ordering of "send" events only. We have shown in Chapter 2 that the two models are not comparable concerning expressive power. Later in this section, we will show that

the realizability analysis for the two model are essentially different. A notion of *well-formedness* is defined in [69] for SLCFSM, which is a necessary condition for freedom of deadlocks and freedom of unspecified receptions. The realizability conditions proposed in this chapter are sufficient conditions for realizability, and for standard FSA conversation protocols, combined with the conditions in Theorem 3.11, these conditions are the sufficient conditions to guarantee freedom of deadlocks, and freedom of unspecified message receptions. It is interesting to note that state space reduction techniques such as fair reachability analysis [56] for CFSM have a similar idea to balance execution steps among machines, and there are sufficient and necessary conditions to identify finite fair reachable state space so that the detection of some specific logic errors such as deadlock and unspecified receptions is decidable. Our results differ from the fair reachability analysis in that we support general LTL model checking, and allow arbitrary interconnection patterns among peers (fair reachability analysis in [56] requires a cyclic shape of interconnection).

In the following, we give a detail comparative study of the realizability analysis for MSC graphs and our work in [39, 37, 40].

3.4.1 Comparison with MSC Graph

In [6, 7], weak and safe realizability problems were raised on MSCs (a finite set of MSCs) and MSC graphs respectively. Alur et al. showed that the decision of realizability for MSCs is decidable, however it is not decidable for MSC graphs. They gave the following sufficient and necessary conditions for the decision of realizability.

1. An MSCs (MSC graph) M is weakly realizable if and only if $L(M)$ is the *complete* and *well-formed* join closure of itself. (Here *complete* means each send event has a matching receive event, and *well-formed* means each receive event has a matching send event.)

2. An MSCs (MSC graph) M is safely realizable if and only if condition 1 is satisfied and $prefix(L(M))$ is the *well-formed* join of itself. Safe realizability is the weak realizability plus deadlock freedom during composition of peers.

These two conditions look very similar to our *lossless join* property. However there are key differences here: 1) in the MSC model, the conditions are sufficient and necessary conditions, while in conversation based model, lossless join is a sufficient condition only, and, 2) it is undecidable to decide the two conditions for MSC graph, because the requirement of well-formed and completeness due to the queue factor. Alur et al. introduces a third condition called *boundedness* condition, which ensures that during the composition of peers the queue length will not exceed a certain preset bound (on the size of the MSC graph). This condition can be very restrictive, for example, Fig. 2.9(c) does not satisfy the boundedness condition because its queue length can be an arbitrary number. Note that the realizability conditions in our conversation model does not require queue length bounded. In addition each of the realizability conditions in conversation model can be checked independently, and the decision procedure is decidable.

In the following, we present another example to illustrate the difference of the realizability conditions between the two models. Fig. 3.7(a) is an equivalent MSC graph of the conversation protocol in Fig. 2.5. Fig. 3.7(b) is an MSC implied by

the MSC graph.

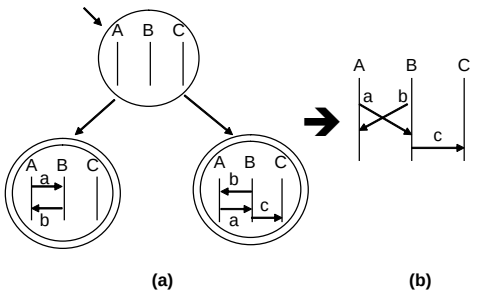


Figure 3.7. The Equivalent MSC graph for Fig. 2.5

Consider how Fig. 3.7(a) and Fig. 2.5 violate the realizability conditions in the two models. Fig. 3.7(a) violates the condition (1) for weakly realizability, because the implied MSC is included in its join closure. However, the conversation protocol at Fig. 2.5 does not violate the *lossless join* condition, rather it violates the autonomy condition. It is not hard to see that the "lossless join" condition in the two models are essentially different.

Chapter 4

Symbolic Realizability and Synchronizability Analyses

In Chapter 3 we studied the problem of *realizability*, i.e., given a conversation protocol, can a web service composition be synthesized to generate behaviors as specified by the protocol. Several sufficient realizability conditions are proposed to ensure realizability. However, the framework presented in Chapter 2 and Chapter 3 is still a step away from practical web service applications, since message contents and data semantics are ignored. It is interesting to ask: can the realizability analysis in Chapter 3 work when data semantics is associated with a conversation protocol? Specifically, given a *GA conversation protocol* which is specified by a Guarded Automaton (GA), where each transition in the GA is equipped with a *guard* to manipulate data. Let its *skeleton* be the FSA generated by removing all data and guards from the GA. One natural question is: if the skeleton is realizable, does it imply that the GA conversation protocol is realizable? This chapter, based on our preliminary results in [42], answers the above

questions. In addition, we extend the work in Chapter 3 to achieve more accurate analysis by considering data semantics. To overcome the state-space explosion caused by the data semantics, we propose a symbolic analysis technique for each realizability condition. In addition, we show that the analysis of the autonomous condition can be achieved using an iterative refinement approach.

This chapter is organized as follows. We first refine the formal specification framework proposed in Chapter 2, to bring in message contents. Then we introduce a light-weight skeleton analysis for GA conversation protocols. Next we present the error-trace guided refined analysis for the autonomous condition, as well as the symbolic analyses for the other two realizability conditions. Finally, we briefly summarize the symbolic synchronizability analysis, and conclude the chapter.

4.1 The Guarded Automata Model

This section extends the automata-theoretic model defined in Chapter 2. As we mentioned earlier, the composition of web services can be specified using either a bottom-up or a top-down approach. In this chapter, both specification approaches are based on the use of Guarded Automata (GA), which allow message classes to have contents and use guards to manipulate data semantics. We begin this section with an extension of the standard composition schema, then we present the technical details of GA conversation protocols and GA web service compositions.

4.1.1 GA Composition Schema

In a composition schema for the new Guarded Automata model, each message is allowed to have contents. Note that in this chapter the organization of message contents is always "flattened"; a version of the GA model with more complex type support such as XML Schema [82] is presented in Chapter 6. Formally, a GA composition schema is defined as below.

Definition 4.1 *A* GA composition schema *is a tuple* (P, M, Σ) *where* (P, M) *is a standard composition schema, where each message class* $c \in M$ *has a finite set of attributes which has a static data type (such as integer, boolean, enumerated and character). Let* $\text{ATTR}(c)$ *and* $\text{DOM}(c)$ *denote the set of attributes and the domain for each message class* $c \in M$*, the* message alphabet Σ *is defined as follows:*

$$\Sigma = \bigcup_{c \in M} \{c\} \times \text{DOM}(c),$$

where each message $m \in \Sigma$ *is an* instance *of a message class in* M*. Similarly, for each* $p_i \in P$*, its* input message alphabet $\Sigma_i^{\text{in}} = \bigcup_{c \in M_i^{in}} \{c\} \times \text{DOM}(c)$*, its* output message alphabet $\Sigma_i^{\text{out}} = \bigcup_{c \in M_i^{out}} \{c\} \times \text{DOM}(c)$*, and let* $\Sigma_i = \Sigma_i^{\text{in}} \cup \Sigma_i^{\text{out}}$.

We use $\text{TYPE}(m)$ to represent the projection of a message m to M and $\text{CON}(m)$ to represent the the projection of message m to $\text{DOM}(\text{TYPE}(m))$. Given a word $w \in \Sigma$, let $w = w_0 w_1 \dots w_k$ where w_j is the j'th message in w, the projection of w on message classes, written as $\pi_{\text{TYPE}}(w)$, is $\text{TYPE}(w_0) \circ \text{TYPE}(w_1) \circ \cdots \circ \text{TYPE}(w_k)$ where "$\circ$" is the concatenation operator.

Example 4.1 The diagram on the left side of Fig. 4.1 defines a GA composition schema for a simplified version of Example 2.1. There are three peers, Store, Bank

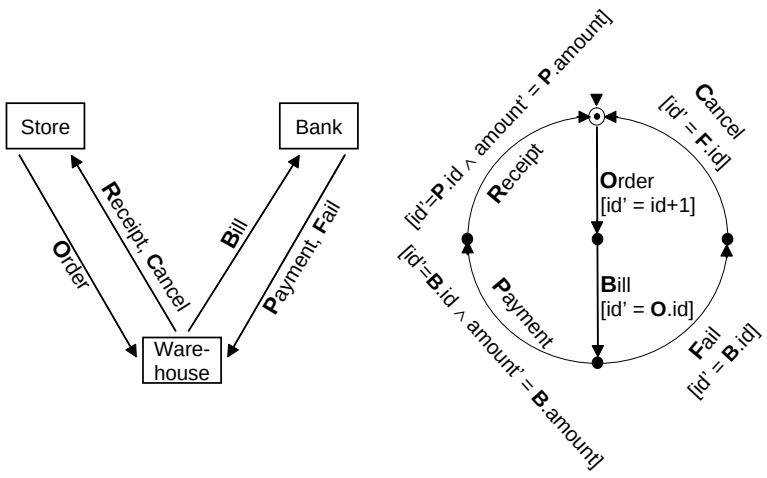


Figure 4.1. A Simplified Warehouse Example

and Warehouse. Message classes, such as **O**rder, **B**ill, and **P**ayment, are transmitted among these three peers. In the rest of this chapter, we assume that message classes **B**ill, **P**ayment and **R**eceipt have two integer attributes id and amount, and the rest of message classes in Fig. 4.1 has one attribute id only. As a message is an instance of a message class, it is written in the form of "class(contents)". For example, $\mathbf{B}(100, 2000)$ stands for a **B**ill whose id is 100 and amount is 2000. Here **B**ill is represented using its capitalized first letter **B**. ∎

4.1.2 GA Conversation Protocol

We now formally define a GA conversation protocol which is specified using a Guarded Automaton.

Definition 4.2 *A* GA conversation protocol *is a tuple* $\langle (P, M, \Sigma), \mathcal{A} \rangle$*, where* (P, M, Σ) *is a GA composition schema, and* $\mathcal{A}$ *is a Guarded Automaton (GA).* $\mathcal{A}$ *is represented using a tuple* $(M, \Sigma, T, s, F, \delta)$ *where* M *and* Σ *are the set of*

message classes and messages respectively, T is a finite set of states, $s \in T$ is the initial state, $F \subseteq T$ is a set of final states, and δ is the transition relation. Each transition $\tau \in \Delta$ is in the form of $\tau = (s, (c, g), t)$, where $s, t \in T$ are the source and the destination states of τ, $c \in M$ is a message class and g is the guard *of the transition.*

Fig. 4.1 presents an example GA conversation protocol where the diagram on the right side is its GA specification. A GA differs from a conventional FSA in its transition guards. During a run of a GA, a transition is taken only if the guard evaluates to true[1]. For example, the guard of the transition to send **Order** is: $\mathbf{Order.id}' = \mathbf{Order.id} + 1$, which intends to increment the value of attribute id by 1 whenever a new **Order** message is sent. Here the primed form of an attribute stands for the "next value" of that attribute, and the non-primed form refers to the "current value". With both primed and non-primed forms of attributes, we can express the semantics of "assignment".

Formally each guard g is a predicate of the following form:

$$g(\text{ATTR}(c'), \text{ATTR}(M_i^{in} \cup M_i^{out}))$$

where $\text{ATTR}(c')$ are the primed attributes of the message that is being sent, and $\text{ATTR}(M_i^{in} \cup M_i^{out})$ are the attributes of the *latest* instances of the message classes that are received or sent by peer p_i, where p_i is the sender of message class c (if for a message class there is no instance received or sent yet, then attribute values for that message class are undefined).

[1]Note that each guard is written as a constraint, which is a frequently used form to describe transition systems in symbolic model checkers (e.g., SMV [18] and Action Language Verifier [17])

Example 4.2 The GA conversation protocol in Fig. 4.1 describes the desired message exchange sequence of the simplified warehouse example: an **o**rder is placed by Store to Warehouse, and then Warehouse sends a **B**ill to the Bank. The Bank either responds with a **P**ayment or rejects with a **F**ail message. Finally Warehouse issues a **R**eceipt or a **C**ancel message. The guards determine the contents of the messages. For example, the id and amount of each **P**ayment must match those of the latest **B**ill message. ∎

Next we formally define the language accepted by a GA. For each message class c, we define $\widehat{\text{DOM}}(c) = \text{DOM}(c) \cup \{\perp\}$ where $\perp$ represents undefined attribute values. A *configuration* of a GA $\mathcal{A}(M, \Sigma, T, s, F, \Delta)$ is a tuple $(t, \vec{m})$ where $t \in T$ and the message vector $\vec{m} \in \widehat{\text{DOM}}(c_1) \times \cdots \times \widehat{\text{DOM}}(c_k)$, $k = |M|$, keeps track of the latest instance of each message class. For a vector $\vec{m}$ and message class c, let $\vec{m}[c]$ denote its projection to $\widehat{\text{DOM}}(c)$. A configuration $(t_1, \vec{m_1})$ is said to *derive* configuration $(t_2, \vec{m_2})$ *via* a message $m \in \Sigma$, written as: $(t_1, \vec{m_1}) \xrightarrow{m} (t_2, \vec{m_2})$ if there is a transition $(t_1, (c, g), t_2) \in \Delta$ such that

- attributes of m and $\vec{m_1}$ satisfy the guard g, i.e., $g(\text{ATTR}(m), \text{ATTR}(\vec{m_1}))$ is true, and
- $\vec{m_2}[\text{TYPE}(m)] = m$, and $\vec{m_1}$ and $\vec{m_2}$ have same instances for all the message classes, i.e., for each $c \in M$ such that $c \neq \text{TYPE}(m)$, $\vec{m_1}[c] = \vec{m_2}[c]$.

Let $\mathcal{A} = (M, \Sigma, T, s, F, \Delta)$ be a GA and $w = w_1 w_2, \ldots, w_n$ be a finite word over Σ, a *run* of $\mathcal{A}$ for w is a finite sequence of configurations $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that 1) $\gamma_0 = (s, (\perp, \ldots, \perp))$ where s is the initial state of $\mathcal{A}$, and 2) for each $0 \leq i < |w|$, $\gamma_i \xrightarrow{w_{i+1}} \gamma_{i+1}$. A word w is accepted by $\mathcal{A}$ if there exists a *complete run* for w s.t. the state of $\mathcal{A}$ at the last configuration is a final state. For example,

it is not hard to infer that one possible word accepted by the GA in Fig. 4.1 is:

$$\mathbf{O}(0), \mathbf{B}(0,100), \mathbf{P}(0,100), \mathbf{R}(0,100), \mathbf{O}(1), \mathbf{B}(1,200), \mathbf{F}(1), \mathbf{C}(1)$$

Given a GA conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$, its language $L(\mathcal{P}) = L(\mathcal{A})$. Note that $L(\mathcal{P})$ is a language over alphabet Σ (instead of M).

Based on the definition of guards, a GA conversation protocol is only able to remember the attributes of the last sent message for each message class. We do not think this is an important restriction based on the web services we studied. Note that, more information about the sent and received messages can be stored in the states of the conversation protocol. Another approach would be to extend the guard definition so that the guards can refer to the last ℓ instances of each message class where ℓ is a fixed integer value. Such an extension would not effect the results we will discuss in the following sections.

4.1.3 GA Web Service Composition

Bottom-up specified GA web service compositions also build upon Guarded Automata, however, the GA used to describe each peer is a bit different than the one used to describe a conversation protocol. The formal definition is given as below.

Definition 4.3 *A* GA web service composition *is denoted using a tuple* $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$*, where* (P, M, Σ) *is the GA composition schema,* $n = |P|$*, and for each* $i \in [1..n]$*:* $\mathcal{A}_i$ *is the peer implementation for* $p_i \in P$*. Each* $\mathcal{A}_i$ *is a tuple* $(M_i, \Sigma_i, t_i, s_i, F_i, \delta_i)$ *where* M_i*,* Σ_i*,* t_i*,* s_i*,* F_i*, and* δ_i *are the set of message classes, set of messages, set of states, initial state, final states, and transition*

relation, respectively. A transition $\tau \in \delta_i$ can be one of the following three types: a send transition $(t, (!\alpha, g_1), t')$, a receive transition $(t, (?\beta, g_2), t')$, and an ϵ-transition $(t, (\epsilon, g_3), t')$, where $t, t' \in T_i$, $\alpha \in M_i^{out}$, $\beta \in M_i^{in}$, and g_1, g_2, and g_3 are predicates in the forms of $g(\text{ATTR}(M_i^{in} \cup M_i^{out}))$, $g(\text{ATTR}(\alpha'), \text{ATTR}(M_i^{in} \cup M_i^{out}))$, and $g(\text{ATTR}(\beta'), \text{ATTR}(M_i^{in} \cup M_i^{out}))$, respectively.

As usual, send and receive transitions are denoted using "!" and "?". In an ϵ-transition, the guard determines if the transition can take place, based on the contents of the latest message for each message class related to that peer. For a receive transition $(t, (?\beta, g), t')$, its guard determines whether the transition can take place based on the contents of the latest messages (i.e., the $\text{ATTR}(M_i^{in} \cup M_i^{out})$ in the formula of g) as well as the message at the queue head (i.e., the $\text{ATTR}(\beta')$). Notice that, even if the message at the queue is of class β, it is possible that the peer gets stuck because the contents of the queue head might not satisfy the predicate g. For a send transition $(t, (!\alpha, g), t')$, the guard g determines not only the transition condition but also the contents of the message being sent (i.e., the $\text{ATTR}(\alpha')$).

Fig. 4.2 shows an example GA composition which realizes the GA conversation protocol in Fig. 4.1. Note that in the figure, a transition without guard actually has a guard "`true`".

Next we present the formal definition of a run and a conversation of a GA composition. Given a GA composition $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, where $n = |P|$ and $k = |M|$, a *global configuration* of $\mathcal{S}$ is a $(2n + 3)$-tuple of the form

$$(Q_1, t_1, ..., Q_n, t_n, w, \vec{s}, \vec{c})$$

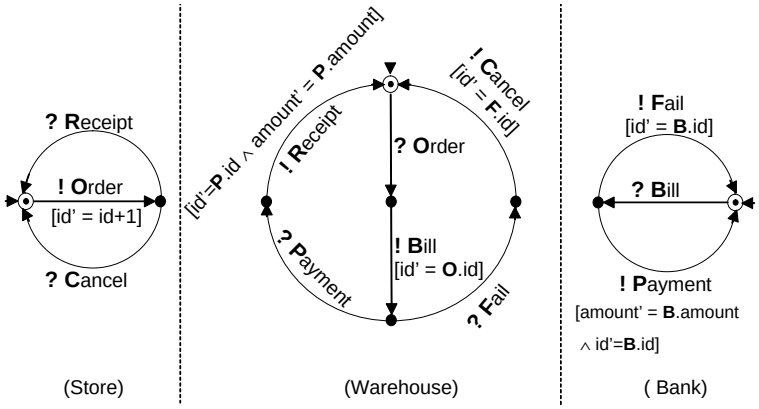


Figure 4.2. A Realization of Fig. 4.1

where for each $j \in [1..n]$, $Q_j \in (\Sigma_j^{\text{in}})^*$ is the queue content of peer p_j, t_j is the state of p_j, $w \in \Sigma^*$ is the global watcher which records the sequence of messages that have been transmitted, and message vectors $\vec{s}, \vec{c} \in \widehat{\text{DOM}}(c_1) \times \cdots \times \widehat{\text{DOM}}(c_k)$ record the latest *sent* and *consumed* instances (resp.) for each message class.

For two configurations $\gamma = (Q_1, t_1, ..., Q_n, t_n, w, \vec{s}, \vec{c})$, and $\gamma' = (Q'_1, t'_1, ..., Q'_n, t'_n, w', \vec{s'}, \vec{c'})$, we say that γ *derives* γ', written as $\gamma \rightarrow \gamma'$, if one of the following holds:

- (Peer p_j executes an ϵ-move) there exists $j \in [1..n]$ such that
  - $(t_j, (\epsilon, g), t'_j) \in \delta_j$, and the predicate g evaluates to `true` on the input messages (of p_j) in $\vec{c}$, and the output messages (of p_j) in $\vec{s}$, and
  - $Q'_j = Q_j$,
  - for each $k \neq j$, $Q'_k = Q_k$ and $t'_k = t_k$, and
  - $w' = w$, $\vec{s'} = \vec{s}$ and $\vec{c'} = \vec{c}$.
- (Peer p_j consumes an input) there exist $j \in [1..n]$ and $a \in \Sigma_j^{\text{in}}$ such that

- $(t_j, (?\text{TYPE}(a), g), t'_j) \in \delta_j$, and the predicate g evaluates to `true` on the contents of the input message a (which is at the input queue head), the input messages (of p_j) in $\vec{c}$, and the output messages (of p_j) in $\vec{s}$, and
- $Q_j = aQ'_j$,
- for each $k \neq j$, $Q'_k = Q_k$ and $t'_k = t_k$, and
- $w' = w$, $\vec{s'} = \vec{s}$. $\vec{c'}[\text{TYPE}(a)] = a$ and $\vec{c'}$ agrees with $\vec{c}$ on all other message classes.

- (Peer p_j sends an output to peer p_k) there exist $j, k \in [1..n]$, a message $b \in \Sigma_j^{\text{out}} \cap \Sigma_k^{\text{in}}$, and predicate g such that
  - $(t_j, (!\text{TYPE}(b), g), t'_j) \in \delta_j$, and the predicate g evaluates to `true` on message b, the input messages (of p_j) in $\vec{c}$, and the output messages (of p_j) in $\vec{s}$, and
  - $Q'_k = Q_k b$, and
  - $Q'_l = Q_l$ for each $l \neq k$, and $t'_l = t_l$ for each $l \neq j$, and
  - $w' = wb$, and $\vec{c'}$=$\vec{c}$, and $\vec{s'}[\text{TYPE}(b)] = b$, and $\vec{s'}$ agrees with $\vec{s}$ on all other message classes.

Based on the above definition of derivation, we can define the concept of conversations. Given a global configuration $\gamma = (Q_1, t_1, ..., Q_n, t_n, w', \vec{s}, \vec{c})$, we denote the value of the global watcher w' in γ as $gw(\gamma) = w'$. A word w is a *conversation* of $\mathcal{S}$ if there exists a *run* $\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_j$ of $\mathcal{S}$ such that

1. $\gamma_0 = (\epsilon, s_1, ..., \epsilon, s_n, \epsilon, [\perp, ..., \perp], [\perp, ..., \perp])$ is the initial configuration, where s_i is the initial state of p_i for each $i \in [1..n]$, and
2. for each $0 \leq i < j$, $c_i \rightarrow c_{i+1}$, and

3. $\gamma_j = (\epsilon, f_1, ..., \epsilon, f_n, w, [...], [...])$ is a final configuration where f_i is a final state of p_i for each $i \in [1..n]$, queue of each peer is empty, and $gw(\gamma_j) = w$.

Let $\mathcal{C}(\mathcal{S})$ denote the set of conversations of a GA composition $\mathcal{S}$. We say $\mathcal{S}$ *realizes* a GA conversation protocol $\mathcal{P}$ if $\mathcal{C}(\mathcal{S}) = L(\mathcal{P})$.

4.2 Cartesian Product and Projection

This section discusses the operations (namely "projection" and "Cartesian product") that convert between GA conversation protocols and GA web service compositions. Different than the standard FSA model, the projection of GA conversation protocols is much harder and trickier. This leads to the extra difficulty in the symbolic realizability analysis for GA conversation protocols discussed later in this chapter.

4.2.1 Cartesian Product

The algorithm of constructing Cartesian product for Guarded Automata is extended from the algorithm for standard FSA, by considering the handling of guards. Let $\mathcal{S} = (\langle (P, M, \sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be a GA web service composition, and for each $i \in [1..n]$, $\mathcal{A}_i$ is represented using tuple $(M_i, \Sigma_i, T_i, s_i, F_i, \delta_i)$. The Cartesian product of all peers in $\mathcal{S}$ is a Guarded Automaton $\mathcal{A}' = (M, \Sigma, T', s', F', \delta')$, where each state $t' \in T'$ is associated with a tuple $(t_1, \ldots, t_n)$ and for each $i \in [1..n]$ t_i is a state of peer $\mathcal{A}_i$. The initial state s' of $\mathcal{A}'$ corresponds to the tuple $(s_1, \ldots, s_n)$, and a final state in F' corresponds to a tuple $(f_1, \ldots, f_n)$ where for each $i \in [1..n]$, f_i is a final state of $\mathcal{A}_i$. Let ρ map each state $t' \in T'$ to the

corresponding tuple, and let $\rho(t')[i]$ denote the i'th element of $\rho(t')$. For any two states t and t' in T', a transition $(t, (m, g'), t')$ is included in δ'_i if there exists two transitions $(t_i, (!m, g_i), t'_i) \in \delta_i$ and $(t_j, (?m, g_j), t'_j) \in \delta_j$ such that

1. **(two peers take transitions simultaneously)** $\rho(t)[i] = t_i$, and $\rho(t')[i] = t'_i$, and $\rho(t)[j] = t_j$, and $\rho(t')[j] = t'_j$, and for each $k \neq i \wedge k \neq j$, $\rho(t')[k] = \rho(t)[k]$, and

2. **(matching of guards)** $g' = g_i \wedge g_j$, and g' is satisfiable.

Clearly, by the above definition, the construction of a Cartesian product for GA can start from the initial state of the product (which corresponds to the tuple of initial states of all peers), then iteratively include new transitions and states. Obviously, the construction can always terminate because the number of transitions and states of all peers is finite. However, note that, different than the Cartesian product construction for FSA, the algorithm here requires the ability to decide the satisfiability of symbolic constraints.

4.2.2 Projection

We now develop algorithms to project a GA conversation protocol to peer prototypes. We first show that, a GA protocol with infinite domains may not always have an "exact" projection. Although for finite domains, an "exact" projection is always possible, the cost of the exact projection is high. We then introduce several "coarse" projections which can save projection cost.

Definition 4.4 *A GA conversation protocol* $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ *is called a* Infinite domain (I-)GA *conversation protocol if* Σ *is an infinite set; otherwise* $\mathcal{P}$ *is*

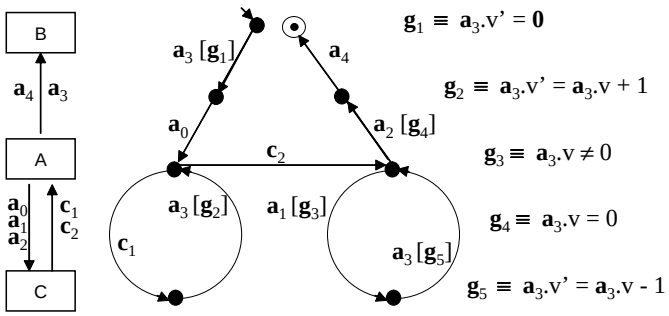


Figure 4.3. The I-GA Conversation Protocol for Example 4.3

an Finite domain (F-)GA *conversation protocol. Similarly, a GA composition is either an F-GA composition or an I-GA composition.*

Exact Projection of GA Conversation Protocols

In the following, we show an I-GA conversation may not have an exact projection.

Example 4.3 Fig. 4.3 presents an I-GA conversation protocol (let it be $\mathcal{P}$) which involves three peers A, B, C. Message class $\mathbf{a}_3$ contains a single infinite-domain integer attribute `v`, and all other message classes do not have message contents. Clearly, only those transitions on $\mathbf{a}_3$ have meaningful guards, other transitions have guards "`true`". The message exchange, as described by $\mathcal{P}$, proceeds as follows. In the initial stage, peer A sends an $\mathbf{a}_3$, to initialize its attribute `v` (used as a counter) to the value 0. Then A informs peer C (with $\mathbf{a}_0$) to start the sending of $\mathbf{c}_1$. For each $\mathbf{c}_1$, peer A increments the counter of $\mathbf{a}_3$ by 1. After message $\mathbf{c}_2$ is received, peer A starts to send back a sequence of $\mathbf{a}_1$ to B (concluded with $\mathbf{a}_2$). Because of the counter attribute in $\mathbf{a}_3$, the number of $\mathbf{a}_1$ is exactly the same as $\mathbf{c}_1$. Hence, for each conversation $w \in L(\mathcal{P})$, its projection to message alphabet,

i.e., $\pi_{\text{TYPE}}(w)$, must be of the following form:

$$\mathbf{a}_3\mathbf{a}_0(\mathbf{c}_1\mathbf{a}_3)^n\mathbf{c}_2(\mathbf{a}_1\mathbf{a}_3)^n\mathbf{a}_2\mathbf{a}_4,$$

where $n \geq 0$. Therefore, the projection of $L(\mathcal{P})$ to peer C, i.e., $\pi_C(L(\mathcal{P}))$ is the set $\{\mathbf{a}_0{\mathbf{c}_1}^n\mathbf{c}_2{\mathbf{a}_1}^n\mathbf{a}_2 \mid n \geq 0\}$, which is a context free language. Notice that, since all messages (sent or received) by peer C does not have message contents, thus any GA on the alphabet of C is essentially a standard FSA, and it is not able to accept $\pi_C(L(\mathcal{P}))$. ∎

The following proposition summarizes the discussion in Example 4.3.

Proposition 4.4 There exists an I-GA conversation protocol $\mathcal{P}$ on some composition schema (P, M, Σ) such that there exists $1 \leq i \leq |P|$ such that for each $\mathcal{A}_i$ on the (message class and message) alphabet (M_i, Σ_i): $L(\mathcal{A}_i) \neq \pi_i(L(\mathcal{P}))$.

We show that for an F-GA conversation protocol $\mathcal{P}$, we can always construct a corresponding projected composition $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ where each peer implementation of $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ is an "exact" projection of $\mathcal{P}$. The projection depends on the following two lemmas (Lemma 4.5, and Lemma 4.6).

Lemma 4.5 For each F-GA $\mathcal{A} = (M, \Sigma, T, s, F, \delta)$, there is a standard FSA $\mathcal{A}'$ on Σ such that $L(\mathcal{A}') = L(\mathcal{A})$.

Proof: We can construct $\mathcal{A}' = (\Sigma, T', s', F', \delta')$ as follows. Let C be the set of all configurations of $\mathcal{A}$. Because Σ is finite, C must be a finite set. We construct the state set T' such that $|T'| = |C|$, and let the one to one and onto mapping ρ maps a configuration $c \in C$ to a state t' in T. For configuration $c_0 = (s, [\perp, \ldots, \perp])$, we

make $\rho(c_0)$ the initial state of $\mathcal{A}'$, and for each configuration $c_i = (f, [..])$ where f is a final state of $\mathcal{A}$, we make $\rho(c_i)$ a final state of $\mathcal{A}'$. For each derivation $c \xrightarrow{m} c'$ of $\mathcal{A}$, we include a transition $(\rho(c), m, \rho(c'))$ in δ'. Obviously, $\mathcal{A}'$ bisimulates every run on the configurations of $\mathcal{A}$, and $L(\mathcal{A}') = L(\mathcal{A})$. ∎

Lemma 4.6 Given a message class alphabet M and the corresponding message alphabet Σ, for each FSA $\mathcal{A}$ on Σ, there is an F-GA $\mathcal{A}'$ such that $L(\mathcal{A}') = L(\mathcal{A})$.

Proof: Let $\mathcal{A} = (\Sigma, T, s, F, \delta)$, we can define $\mathcal{A}' = (M, \Sigma, T', s', F', \delta')$ as follows: $T' = T$, $s' = s$, $F' = F$, $|\delta| = |\delta'|$, and for each transition $(t, m, t') \in \delta$ there exists a corresponding $(t, (\text{TYPE}(m), g), t') \in \delta'$ such that g is the predicate which assigns the value of m to the the message being sent in the transition. ∎

Lemma 4.5 and Lemma 4.6 immediately implies the following theorem.

Theorem 4.7 For each F-GA conversation protocol $\mathcal{P}$, there exists a F-GA web service composition $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ where for each $i \in [1..n]$: $L(\mathcal{A}_i) = \pi_i(L(\mathcal{P}))$.

Proof: Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ where $n = |P|$. Since Σ is finite, construct the corresponding FSA $\mathcal{A}'$ for $\mathcal{A}$ such that $L(\mathcal{A}') = L(\mathcal{A})$, according to Lemma 4.5. Then project $\mathcal{A}'$ to each peer prototype, and let them be $\mathcal{A}'_1$, ..., $\mathcal{A}'_n$, respectively. Convert each $\mathcal{A}'_i$ into an F-GA $\mathcal{A}_i$ by Lemma 4.6, and we get the F-GA web service composition $\langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$. ∎

Definition 4.5 *Given an F-GA conversation protocol $\mathcal{P}$, let $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be the corresponding GA composition given in the proof of Theorem 4.7. $\mathcal{S}$ is called the* (exact) projected composition *of $\mathcal{P}$, written as $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$.*

Procedure ProjectGA($\langle (P, M, \Sigma), \mathcal{A} \rangle$, i): GA

Begin

Let $\mathcal{A}' = (M, \Sigma, T, s_0, F, \delta)$ be a copy of $\mathcal{A}$.

Substitute each $(q_1, (a, g_1), q_2)$ where $a \notin M_i$ with $(q_1, (\epsilon, g'_1), q_2)$.

Substitute each $(q_1, (a, g_2), q_2)$ where $a \in M_i^{in}$ with $(q_1, (?a, g'_2), q_2)$.

Substitute each $(q_1, (a, g_3), q_2)$ where $a \in M_i^{out}$ with $(q_1, (!a, g'_3), q_2)$.

// The generation of g'_1, g'_2 and g'_3 uses either **Coarse Processing 1** or **Coarse Processing 2**.

// **Coarse Processing 1:**

// $g'_1 = g'_2 =$"true", and $g'_3 = g_3$.

// **Coarse Processing 2:**

// g'_1, g'_2, g'_3 are the predicates generated from g_1, g_2, g_3 (resp.),

// by eliminating non-related message attributes via existential quantification.

// Obviously, $g'_3 = g_3$.

return $\mathcal{A}'$. **End**

Figure 4.4. Coarse Projection of a GA Conversation Protocol

Coarse Projection

The construction of $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ (as shown in Theorem 4.7) is very costly – it requires essentially a reachability analysis of the state space of the F-GA conversation protocol. In Fig. 4.4, we present a light-weight however not "exact" projection algorithm, which works for both F-GA and I-GA conversation protocols.

The coarse projection algorithm in Fig. 4.4 simply replace each non-related transition with ϵ-transitions, and add "!" and "?" for send and receive transitions respectively. There are two different levels of "coarse processing" that can be used in the algorithm. In **Coarse Processing 1**, the guards of ϵ-transitions and receive transitions are essentially dropped (by setting them to "true"), and the guards of send transitions remain the same. In **Coarse Processing 2**, we

use existential quantification to eliminate non-related message attributes from the formula of the guard, and this produces a "more accurate" projection than **Coarse Processing 1**.

Example 4.8 Given a GA conversation protocol on three peers p_1, p_2 and p_3. Let $\tau = (t, (m_1, g), t')$ be a transition in the protocol, where $m_1 \in M_3^{out} \cap M_1^{in}$, $m_2 \notin M_1$, and

$$g \equiv m_1.\mathsf{id} + m_2.\mathsf{id} < 3 \ \wedge \ m_2.\mathsf{id} > 0.$$

We suppose $m_1.\mathsf{id}$ and $m_2.\mathsf{id}$ are both of integer type. During the projection to peer p_1, if τ is being processed using **Coarse Processing 1**, the corresponding transition would be $(t, (?m_1, true), t')$; if **Coarse Processing 2** is used, the corresponding transition would be $(t, (?m_1, g'), t')$ where $g' \equiv \exists_{m_2.\mathsf{id}}\ g$, and after simplification, $g' \equiv m_1.\mathsf{id} < 2$. ∎

Definition 4.6 *Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ be a GA conversation protocol. Its* coarse-1 projected GA composition *(written as $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$), is a tuple $\langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ where $n = |P|$, and for each $i \in [1..n]$: $\mathcal{A}_i$ is the result of applying* $\mathsf{ProjectGA}(\mathcal{A}, i)$ *using Coarse Processing 1. Similarly, the* coarse-2 projected GA composition *of $\mathcal{P}$, written as $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$, is the result of applying* $\mathsf{ProjectGA}$ *using Coarse Processing 2.*

Lemma 4.9 Given a GA conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$, and its projections $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$. For each $1 \leq i \leq |P|$, let $\mathcal{A}_i$ and $\mathcal{A}_i'$ be the peer implementation of p_i in $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}1}$ and $\mathcal{S}_{\mathcal{P}}^{\mathrm{PROJ},\mathbb{C}2}$ (resp.), the following is true: $\pi_i(L(\mathcal{P})) \subseteq L(\mathcal{A}_i') \subseteq L(\mathcal{A}_i)$.

1. **Procedure** ElimGA ($\mathcal{A}$): GA
2. **Begin**
3. **Let** $\mathcal{A}' = (M, \Sigma, T, s, F, \delta)$ be a copy of $\mathcal{A}$.
4. **For each** $t \in T$ **Do**
5. include each t' that is reachable from t via ϵ-paths into ϵ-closure(t).
6. **End For**
7. // let $\Upsilon(t, t')$ be the set of non-redundant ϵ-paths from t to t'.
8. // each transition in δ appears at most once in a non-redundant path.
9. // let cond(ℓ) be the conjunction of all guards along a non-redundant path ℓ.
10. **For each** transition $(t, (m, g), t') \in \delta$ **do**
11. include into δ a transition $(t, (m, g'), t'')$ for each t'' in ϵ-closure(t'),
12. where $g' = g \wedge g''$ and $g'' = \bigvee_{\ell \in \Upsilon(t', t'')} \text{cond}(\ell)$.
13. **End For**
14. eliminate all ϵ-transitions from δ.
15. **return** $\mathcal{A}'$.
16. **End**

Figure 4.5. ϵ-transitions Elimination for Guarded Automata

Proof: For each transition $(t, (m, g), t')$, let g_1 and g_2 be the corresponding guards generated by Coarse Processing 1 and Coarse Processing 2. Obviously, $g \Rightarrow g_2 \Rightarrow g_1$, where " $\Rightarrow$ " is the boolean operator "imply". This fact immediately leads to the lemma. ∎

4.2.3 Determinization of Guarded Automata

We now introduce a "determinization" algorithm for Guarded Automata, which is useful in the decision procedures for realizability conditions. The "determinization", as usual, consists of two steps: (1) eliminate ϵ-transitions from a Guarded Automaton, and (2) determinize the resulting GA of step (1). The

1. **Procedure** DeterminizeGA($\mathcal{A}$): GA
2. **Begin**
3. **Let** $\mathcal{A}' = (M, \Sigma, T, s, F, \delta)$ be a copy of ElimGA($\mathcal{A}$).
4. Mark all states in T as "unprocessed".
5. **For each** "unprocessed" state $t \in T$ **Do**
6. **For each** message class $m \in M$ **Do**
7. Let $\{\tau_1, \ldots, \tau_k\}$ include each transition $\tau_i = (t, (m, g_i), t'_i)$ which starts from t and sends m.
8. mark each τ_i as "toRemove"
9. **For each** $c = \ell_1 \wedge \cdots \wedge \ell_k$ where ℓ_i is g_i or $\bar{g_i}$ **Do**
10. **If** c is satisfiable **Then**
11. **Let** s' be a new state name, include s' in T.
12. include $(t, (m, c), s')$ in δ.
13. **For each** $j \in [1..k]$ s.t. g_j (instead of $\bar{g_j}$) appears in c **Do**
14. **For each** transition $\tau' = (t'_j, (m', g'), t''_j)$ from t'_j **do**
15. include $(s', (m', g'), t''_j)$ in δ
16. mark τ' as "toRemove". mark t'_j as "toRemove".
17. **End Do**
18. **End Do**
19. **End If**
20. **End Do**
21. **End Do**
22. remove all states and transitions that are marked as "toRemove".
23. **End Do**
24. return $\mathcal{A}'$.

End

Figure 4.6. Determinization of Guarded Automata

ϵ-transition elimination algorithm is presented in Fig. 4.5, which is an extension of the ϵ-transition elimination for standard FSA.

One interesting part of the algorithm in Fig. 4.5 is the collection and handling of guards (lines 10 to 13). Note that the transition $(t, (m, g'), t'')$ in line 11 is a replacement for a set of paths, where each path is a concatenation of the transition $(t, (m, g), t')$ and an ϵ-path from t' to t''. It is not hard to see that

the guard g' should be the conjunction of g and g'' where g'' is the disjunction of the conjunctions of guards along each ϵ-path from t' to t''. Note that for each ϵ-transition, its guard is only a "transition condition" which does not affect the message instance vector in a GA configuration. Hence it suffices to consider those non-redundant paths, and obviously the number of non-redundant paths is finite. Thus the algorithm in Fig. 4.5 can always terminate.

Fig. 4.6 presents the determinization algorithm for a Guarded Automaton. Note that the idea of the algorithm is rather different than the determinization algorithm for a standard FSA. The key idea of the algorithm is the part (lines 9 to 20), where for each state and each message class, we collect all transitions for that message class, enumerate every combination of guards, and generate a new transition for that combination. For example, suppose at some state t, two transitions are collected for message class m, let their guards be g_1 and g_2 respectively. Four new transitions will be generated for $g_1 \wedge g_2$, $g_1 \wedge \bar{g_2}$, $\bar{g_1} \wedge g_2$, and $\bar{g_1} \wedge \bar{g_2}$ (resp.), and the two original transitions are removed from the transition relation. It is not hard to see that for each word $w \in L(\mathcal{A}')$, where $\mathcal{A}'$ is the resulting GA, there exists one and only one run for w, due to the enumeration of the combinations of guards. Since the procedure of enumerating guards and reassembling states does not introduce or remove any pair of configurations (c,c') s.t. $c \rightarrow c'$, each $\mathcal{A}$ is equivalent to its determinization $\mathcal{A}'$ (after applying DeterminizeGA), i.e., $L(\mathcal{A}) = L(\mathcal{A}')$.

4.3 Revisit Realizability

This section pays a short revisit to the realizability conditions for F-GA conversation protocols, and presents some preliminary results on F-GA conversation protocols.

Definition 4.7 *A GA conversation protocol $\mathcal{P}$ satisfies the* lossless join *condition if $L(\mathcal{P}) = \text{JOINC}(L(\mathcal{P}))$.*

Definition 4.8 *Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ be a GA conversation protocol, and $n = |P|$. $\mathcal{P}$ is said to be* synchronous compatible *if for each word $w \in \Sigma^*$ and each message $\alpha \in \Sigma_a^{\text{out}} \cap \Sigma_b^{\text{out}}$ for $a, b \in [1..n]$, the following holds:*

$$(\forall i \in [1..n],\ \pi_i(w) \in \pi_i(L_{\preceq}^*(\mathcal{A}))) \ \wedge\ \pi_a(w\alpha) \in \pi_a(L_{\preceq}^*(\mathcal{A}))$$
$$\Rightarrow \quad \pi_b(w\alpha) \in \pi_b(L_{\preceq}^*(\mathcal{A})).$$

Definition 4.9 *A GA conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ is* autonomous *if for each peer prototype $p_i \in P$ and for each finite prefix $w \in L_{\preceq}^*(\mathcal{A})$, p_i at $\pi_i(w)$ is exactly one of the following: output-ready, input-ready, or terminate-ready.*

We now concentrate on the realizability analysis for GA conversation protocols with finite domains.

Lemma 4.10 Deciding if an F-GA conversation protocol $\mathcal{P}$ satisfies the lossless join, synchronous compatible and autonomous conditions is EXPTIME on the size of the equivalent FSA conversation protocol of $\mathcal{P}$.

Proof: Convert $\mathcal{P}$ to the equivalent FSA conversation protocol $\mathcal{P}'$ according to Lemma 4.5. Obviously, $\mathcal{P}$ is lossless join, if and only if $\mathcal{P}'$ is lossless join,

and similar observations apply to other two conditions. The decision of these realizability conditions on $\mathcal{P}'$ takes EXPTIME on its size. ∎

Theorem 4.11 An F-GA conversation protocol $\mathcal{P}$ is realizable (by $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$) if the lossless join, synchronous compatible and autonomous conditions are satisfied.

Proof: Convert $\mathcal{P}$ to the equivalent FSA conversation protocol $\mathcal{P}'$ according to Lemma 4.5. Obviously, when the three realizability conditions are satisfied, $\mathcal{P}'$ is realized by its projected composition, which can be converted to $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$. ∎

Unfortunately, we currently do not have similar results for I-GA conversation protocols. We suspect that even if an I-GA conversation protocol satisfies these three realizability conditions (where the decision problem for these three conditions may not even be decidable), there may not exist a finite control state system which realizes the protocol.

4.4 Skeleton Analysis

This section investigates the feasibility of deciding if a GA conversation protocol is realizable by checking its abstract control flows (called skeleton), without considering its data semantics. Note that the skeleton analysis works for both F-GA and I-GA conversation protocols.

Definition 4.10 *Given a GA* $\mathcal{A} = (M, \Sigma, T, s, F, \delta)$*, its skeleton, denoted as* skeleton($\mathcal{A}$)*, is a standard FSA* (M, T, s, F, δ') *where* δ' *is obtained from* δ *by replacing each transition* $(s, (c, g), t)$ *with* (s, c, t)*.*

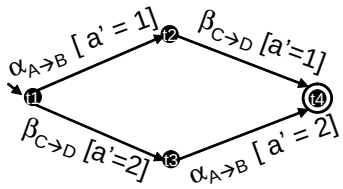


Figure 4.7. The GA Conversation Protocol for Example 4.12

Note that $L(\text{skeleton}(\mathcal{A})) \subseteq M^*$, while $L(\mathcal{A})$ is a subset of Σ^*. For a GA conversation protocol $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$, we can always construct an FSA conversation protocol $\langle (P, M), \text{skeleton}(\mathcal{A}) \rangle$. We call this protocol the *skeleton protocol* of $\mathcal{P}$.

4.4.1 Theoretical Observations

Now, one natural conjecture is: if the skeleton protocol of a GA conversation protocol is realizable, does this imply that the GA protocol is realizable? In Fig. 4.7 we give a counter example.

Example 4.12 The GA conversation protocol shown in Fig. 4.7 has four peers A, B, C, D. There are two message classes in the system: α from A to B and β from C to D. Both message classes have an attribute `a`. The protocol specifies two possible conversations $\alpha(1)\beta(1)$, and $\beta(2)\alpha(2)$. Obviously, the skeleton protocol, which specifies the desired conversation set $\{\alpha\beta, \beta\alpha\}$, is realizable because it satisfies all the three realizability conditions. However, the GA protocol itself is not realizable, because any implementation that generates the specified conversations will also generate the conversation $\beta(1)\alpha(1)$ as well. ∎

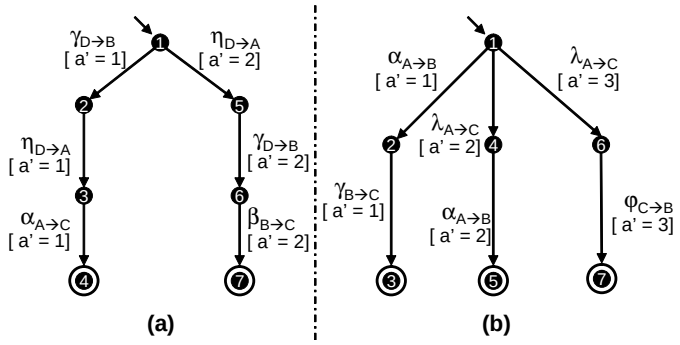


Figure 4.8. The GA Conversation Protocols for Examples 4.13 and 4.14

Example 4.13 Fig. 4.7 is an example where the skeleton is lossless join, however the GA conversation protocol is not. Fig. 4.8(a) is an example where the protocol is lossless join, while its skeleton is not. There are four peers A, B, C, D in Fig. 4.8(a), and all message classes contain a single attribute `a`. In the beginning, peer D informs peer A and B about which path to take by the value of attribute `a` (1 for left branch or 2 for right branch). Then A and B knows who is going to send the last message (α or β), so there is no ambiguity. It can be verified that the protocol is lossless join. However the skeleton of Fig. 4.8(a) is obviously not lossless join, because $\eta\gamma\alpha$ is included in its join closure. ∎

Example 4.14 If we make the message β in Fig. 4.7 from peer C to A, the modified Fig. 4.7 is an example which is not synchronous compatible, yet its skeleton is synchronous compatible. Fig. 4.8(b) is another example, where the GA conversation protocol is actually synchronous compatible however its skeleton is not, because after the partial conversation $\lambda\alpha$, peer B is ready to send γ however peer C is not receptive to it. ∎

The following propositions summarizes Examples 4.12, 4.13, and 4.14.

Proposition 4.15 A GA conversation protocol may be realizable while its skeleton protocol is not realizable.

Proposition 4.16 A GA conversation protocol may not be realizable when its skeleton protocol satisfies the lossless join, synchronous compatible, and autonomous conditions.

Proposition 4.17 A GA conversation protocol may not be realizable while its skeleton protocol is realizable.

Proposition 4.18 A GA conversation protocol may be lossless join while its skeleton protocol is not lossless join.

Proposition 4.19 A GA conversation protocol may not be lossless join while its skeleton protocol is lossless join.

Proposition 4.20 A GA conversation protocol may be synchronous compatible while its skeleton protocol is not synchronous compatible.

Proposition 4.21 A GA conversation protocol may not be synchronous compatible join while its skeleton protocol is synchronous compatible.

The above propositions suggest that we cannot tell if a GA conversation protocol is realizable or not, based on the result of its skeleton protocol. Similar observations apply to the lossless join and synchronous compatible conditions; however, as we will show later, the autonomy of a GA conversation protocol can be implied by the autonomy of its skeleton protocol.

4.4.2 Skeleton Analysis

We now introduce a fourth realizability condition to restrict a GA conversation protocol so that it can be realized by $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$, $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}1}$, and $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}2}$.

Definition 4.11 *Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ be a GA conversation protocol where $\mathcal{A} = (M, \Sigma, T, s, F, \delta)$. $\mathcal{P}$ is said to satisfy the* deterministic guards condition *if for each pair of transitions $(t_1, (m_1, g_1), t_1)'$ and $(t_2, (m_2, g_2), t_2')$, g_1 must be equivalent to g_2, when the following conditions hold:*

1. *$m_1 = m_2$, and*
2. *Let p_i be the sender of m_1. There exists two words $w \in L(\mathcal{A}^*)$ and $w' \in L(\mathcal{A}^*)$ where a partial run of w reaches t_1, and a partial run of w' reaches t_2, and $\pi_i(\pi_{\text{TYPE}}(w)) = \pi_i(\pi_{\text{TYPE}}(w'))$.*

Intuitively, the deterministic guards condition requires that for each peer, according to the GA conversation protocol, when it is about to send out a message, the guard that is used to compute the contents of the message is uniquely decided by the sequence of message classes (note: not messages) exchanged by the peer in the past.

The decision procedure for the deterministic guards condition proceeds as follows: given a GA conversation protocol $\mathcal{P}$, obtain its coarse-1 projected composition $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}1}$. Let $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}1} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$. For each $i \in [1..n]$, regard $\mathcal{A}_i$ as a standard FSA, and get its equivalent deterministic FSA (let it be $\mathcal{A}_i'$). Now each state t in $\mathcal{A}_i'$ corresponds to a set of states in $\mathcal{A}_i$, and let it be represented by $T(t)$. We examine each state t in $\mathcal{A}_i'$, for each message class

$c \in M$, we collect the guards of the transitions that start from a state in $T(t)$ and send message class c. We require that all guards collected for a state/message class pair (t,c) should be equivalent.

Example 4.22 The GA conversation protocol in Fig. 4.7 violates the deterministic guards condition, because (intuitively) peer A has two different guards when sending out α at the initial stage. Formally, to show that the deterministic guards condition is violated, we can find two transitions $(t_1, (\alpha, [a' = 1]), t_2)$ and $(t_3, (\alpha, [a' = 2]), t_4))$, and two words $w = \epsilon$ and $w' = \beta(2)$. Because a run of w reaches $t1$, a run of w' reaches $t3$, and $\pi_A\pi_{\text{TYPE}}(w) = \pi_A\pi_{\text{TYPE}}(w') = \epsilon$, by Definition 4.11, the guards of the two transitions should be equivalent. However, they are not equivalent, and this leads to the violation of the deterministic guards condition. ∎

Theorem 4.23 A GA conversation protocol $\mathcal{P}$ is realized by $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}1}$, $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}2}$, and $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$ if it satisfies the deterministic guards condition, and its skeleton protocol satisfies the lossless join, synchronous compatible and autonomous conditions.

Proof: Let $\mathcal{S}_{\mathcal{P}}^{\text{D,PROJ},\mathbb{C}1}$ be the web service composition generated from $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}1}$ by determinizing each peer and collecting guards as in the check of the deterministic guards condition. We show that $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{D,PROJ},\mathbb{C}1})$. Other conclusions such as $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}2})$ and $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{PROJ}})$ can be directly inferred from Lemma 4.9 and the $L(\mathcal{P}) = \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\text{D,PROJ},\mathbb{C}1})$ we are about to prove.

First, we argue that if the skeleton protocol satisfies the synchronous compatible and autonomous conditions, then during any (complete or partial) run of $\mathcal{S}_{\mathcal{P}}^{\text{D,PROJ},\mathbb{C}1}$, each message is consumed "eagerly", i.e., when the input queue

is not empty, a peer never sends out a message or terminates. This argument can be proved by contradiction. Suppose there is a partial run against this argument, we can find a corresponding partial run of the skeleton composition of $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}}$ (which consists of the skeletons of each peer of $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}}$) where a message class is not consumed eagerly. Now since the skeleton composition satisfy the synchronous compatible and autonomous conditions, and each skeleton peer is a DFSA, by Theorem 3.13, each message class is consumed eagerly, which leads to the contradiction. Therefore our argument should be true.

Now it suffices to show that $\mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}}) \subseteq L(\mathcal{P})$, as $L(\mathcal{P}) \subseteq \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}})$ is obvious. Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A} \rangle$ and let $\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$. Given a word $w \in \mathcal{C}(\mathcal{S}_{\mathcal{P}}^{\mathrm{D,PROJ,\mathbb{C}1}})$, and γ be the corresponding run, we can always construct a run γ' of $\mathcal{A}$ to recognize w. Since $\pi_i(w)$ is accepted by each peer $\mathcal{A}_i$, $\pi_i(\pi_{\mathrm{TYPE}}(w))$ is accepted by skeleton$(\mathcal{A}_i)$. Because skeleton$(\mathcal{A})$ is lossless join, it follows that $\pi_{\mathrm{TYPE}}(w)$ is accepted by skeleton$(\mathcal{A})$, and let $\mathcal{T} : \tau_1 \tau_2 \ldots \tau_{|w|}$ be the path of skeleton$(\mathcal{A})$ traversed to accept $\pi_{\mathrm{TYPE}}(w)$. Since each transition in skeleton$(\mathcal{A})$ is the result of dropping the guard of a corresponding transition, we can have a corresponding path $\mathcal{T}'$ in $\mathcal{A}$. Now we can construct the run $\gamma' = c_0 c_1 \ldots c_{|w|}$, along the path $\mathcal{T}'$. Obviously $c_0 = (s, [\perp\perp ... \perp])$. For each $1 \leq i \leq |w|$, $c_i = (t, \vec{m})$ where t is the i'th state along the path $\mathcal{T}'$, and let $c_i' = (Q_1, t_1, \ldots, Q_n, t_n, \vec{s}, \vec{c})$ be the configuration in γ right before the send of message w_i. The contents of $\vec{m}$ is taken from configuration c_i' such that $\vec{m} = \vec{s}$. Note that, when w_i is sent, its sender compute the contents of w_i based on the information of $\vec{s}$, because all input messages have been consumed (due to the the eager message consumption property we have proved). The above ensures that the simulation of γ can always proceed correctly until the word is accepted. ∎

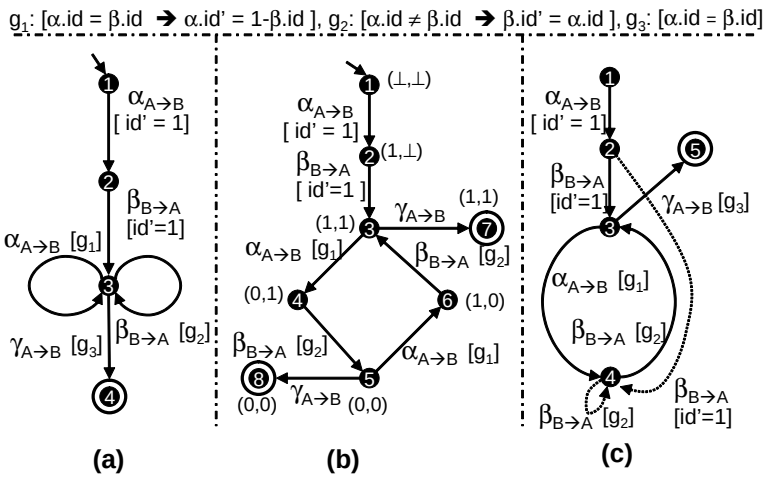


Figure 4.9. Alternating Bit Protocol

Based on Theorem 4.23, we have a light-weight realizability analysis for GA conversation protocols. We check the first three realizability conditions on the skeleton of a conversation protocol (i.e, without considering the guards), and then examine the fourth realizability condition by syntactically check the guards (but actually without analyzing their data semantics).

4.5 Symbolic Analysis

Skeleton analysis may not be very precise. Fig. 4.9(a) is a realizable alternating bit protocol, where its skeleton analysis fails. The GA conversation protocol shown in Fig. 4.9(a) consists of two peers A and B. Message class α is an request, and message class β is an acknowledgment. Both message classes contain an "`id`" attribute. Message class γ is the end-conversation notification. The protocol states that the `id` attribute of requests from A alternates between 0 and 1, and every acknowledgment β must match the `id`.

Let $\mathcal{A}_a, \mathcal{A}_b, \mathcal{A}_c$ be the three conversation protocols shown in Fig. 4.9. It is not hard to see that, the projection of skeleton($\mathcal{A}_a$) to peer A does not satisfy the autonomy condition, because at state 3, there are both input and output transitions. However, $\mathcal{A}_a$ is actually autonomous. If we explore each configuration of $\mathcal{A}_a$, we get $\mathcal{A}_b$, the "equivalent" conversation protocol of $\mathcal{A}_a$. The pair of values associated with each state in $\mathcal{A}_b$ stands for the `id` attribute of α and β. It is obvious that $\mathcal{A}_b$ satisfies the autonomy condition, and hence $\mathcal{A}_a$ should satisfy autonomy as well. In fact to prove that $\mathcal{A}_a$ is autonomous we do not even have to explore each of its configurations like $\mathcal{A}_b$. As we will show later, it suffices to show $\mathcal{A}_c$ is autonomous. Finally notice that $L(\mathcal{A}_a) = L(\mathcal{A}_b) = L(\mathcal{A}_c)$.

4.5.1 Iterative Refined Analysis of Autonomy

The examples in Fig. 4.9 motivates a refined analysis for the autonomous condition: given a conversation protocol $\mathcal{A}$, we can first check its skeleton. If the skeleton analysis fails, we can refine the protocol (e.g. refine $\mathcal{A}_a$ and get $\mathcal{A}_c$), and apply the skeleton analysis on the refined protocol. We can repeat this procedure until we reach the most refined protocol which actually plots the transition graph of the configurations of the original protocol (such as $\mathcal{A}_b$ to $\mathcal{A}_a$). In the following, we first present the theoretical background of the iterative refined analysis.

Theoretical Background

Our refined analysis of autonomy is based on the notion of *simulation*, which is defined as below. A transition system is a tuple (M, T, s, Δ) where M is the set of labels, T is the set of states, s the initial state, and Δ the transition relation.

Generally a transition system can be regarded as an FSA (or an infinite state system) without final states. On the other hand, a standard FSA (M, T, s, F, Δ) can be regarded as a transition system of (M, T, s, Δ); and a GA $(M, \Sigma, T, s, F, \Delta)$ can be regarded as a transition system of the form $(\Sigma, T', s', \Delta')$ where T' contains all configurations of the GA, and Δ' defines the derivation relation between configurations.

Definition 4.12 *A transition system $\mathcal{A}' = (M', T', s', \Delta')$ is said to* simulate *another $\mathcal{A} = (M, T, s, \Delta)$, written as $\mathcal{A} \preceq \mathcal{A}'$, if there exists a mapping $\rho : T \to T'$ and $\varrho : M \to M'$ such that for each (s, m, t) in Δ there is a $(\rho(s), \varrho(m), \rho(t))$ in Δ'. Two transition systems $\mathcal{A}$ and $\mathcal{A}'$ are said to be* equivalent*, written as $\mathcal{A} \simeq \mathcal{A}'$, if $\mathcal{A} \preceq \mathcal{A}'$ and $\mathcal{A}' \preceq \mathcal{A}$.*

Example 4.24 For the three conversation protocols $\mathcal{A}_a, \mathcal{A}_b, \mathcal{A}_c$ in Fig. 4.9, the following is true:

$$\text{skeleton}(\mathcal{A}_b) \preceq \text{skeleton}(\mathcal{A}_c) \preceq \text{skeleton}(\mathcal{A}_a)$$

For example, in the simulation relation $\text{skeleton}(\mathcal{A}_c) \preceq \text{skeleton}(\mathcal{A}_a)$, ρ maps states $1, 2, 3, 4, 5$ in $\text{skeleton}(\mathcal{A}_c)$ to states $1, 2, 3, 3, 4$ of $\text{skeleton}(\mathcal{A}_a)$ respectively, and ϱ is the identity function which maps each message class to itself. For another example, $\mathcal{A}_a \preceq \text{skeleton}(\mathcal{A}_a)$, and $\mathcal{A}_a \simeq \mathcal{A}_b \simeq \mathcal{A}_c$. ∎

Intuitively when $\mathcal{A} \preceq \mathcal{A}'$, each word accepted by $\mathcal{A}$ has a corresponding word accepted by $\mathcal{A}'$, and $\mathcal{A}'$ can contain "more" words than $\mathcal{A}$. It is not hard to infer the following lemma.

Lemma 4.25 For any GA $\mathcal{A}$, $\mathcal{A} \preceq \text{skeleton}(\mathcal{A})$.

Proof: We can construct the mappings from $\mathcal{A}$ to its skeleton. Since a configuration of $\mathcal{A}$ is of the form $(t, \vec{m})$ where t records the local state and $\vec{m}$ is a vector of message instances for each message class. For each configuration $c = (t, \vec{m})$, $\rho(c) = t$, and for each message m, $\varrho(m) = \text{TYPE}(m)$. ∎

Lemma 4.26 For each GA $\mathcal{A} = (M, \Sigma, T, s, F, \Delta)$ on a finite alphabet Σ, there is a standard FSA on alphabet Σ such that $\mathcal{A} \simeq \mathcal{A}'$.

Proof: This lemma directly follows Lemma 4.5, and $\mathcal{A}'$ is the equivalent FSA, as defined in Lemma 4.5. ∎

Theorem 4.27 If $\mathcal{A} \preceq \mathcal{A}'$ and $\mathcal{A}'$ is autonomous, then $\mathcal{A}$ is autonomous.

Proof: We prove by contradiction. Assume that $\mathcal{A}'$ is autonomous but $\mathcal{A}$ is not autonomous, hence we can always a word $w \in L(\mathcal{A}^*)$ which leads to the violation of autonomy (e.g. there exist input message class α and output message class β such that $w\alpha \in L(\mathcal{A}^*)$ and $w\beta \in L(\mathcal{A}^*)$). Now $\rho(w)$ is a word which leads to the violation of autonomy in $\mathcal{A}'$. ∎

Lemma 4.25 and Theorem 4.27 immediately leads to the following.

Corollary 4.28 A GA is autonomous if its skeleton is autonomous.

The Iterative Analysis Algorithm

Based on Corollary 4.28 we have an error-trace guided symbolic analysis algorithm (procedure AnalyzeAutonomy in Fig. 4.10).

Procedure AnalyzeAutonomy($\mathcal{A}$): List

Begin

$\mathcal{A}'$ = DeterminizeGA($\mathcal{A}$)

While true **do**

If skeleton of $\mathcal{A}'$ is autonomous **Then** return null

Find a pair $(s,(m_1,g_1),t_1),(s,(m_2,g_2),t_2)$ violating the autonomy

($\mathcal{A}'$,trace) = Refine($\mathcal{A}'$, $(s,(m_1,g_1),t_1)$, $(s,(m_2,g_2),t_2)$)

If trace $\neq$ null **Then** return trace

End While

End

Figure 4.10. Iterative Analysis

If the input GA is autonomous, AnalyzeAutonomy returns `null`; otherwise it returns the error trace which is a list of configurations. AnalyzeAutonomy starts from the input GA, and refines incrementally. During each cycle, procedure AnalyzeAutonomy analyzes the skeleton of the current GA $\mathcal{A}'$. If the skeleton is autonomous, by Corollary 4.28, AnalyzeAutonomy simply returns and reports that the input GA is autonomous; otherwise, AnalyzeAutonomy identifies a pair of input/output transitions which start from the same state and lead to the violation the autonomy. For example, when analysis is applied to the skeleton of Fig. 4.9(a), the two transitions starting at state 3 will be identified. Then procedure Refine is invoked to refine the current GA. This refinement process continues until the input GA is proved to be autonomous or an concrete error trace is found.

Procedure Refine($\mathcal{A}$, $(s,(m_1,g_1),t_1)$, $(s,(m_2,g_2),t_2)$) : (GA, List)

Begin

If $(\mathsf{Pre}(g_1) \wedge \mathsf{Pre}(g_2))$ is satisfiable **then**

Path = FindPath($\mathcal{A}, s, \mathsf{Pre}(g_1) \wedge \mathsf{Pre}(g_2)$)

If Path $\neq$ null **then** return (null,Path)

End If

Let $\mathcal{A}' = (M', T', s'_0, F', \Delta')$ be a copy of $\mathcal{A}$

$T' = T' - \{s\} + \{s_1, s_2\}, F' = F' - \{s\} + \{s_1, s_2\}$ if $s \in F'$

Substitute each $(t,(m_j,g_j),s)$ in Δ' with

$(t,(m_j,g_j),s_1)$ and $(t,(m_j,g_j),s_2)$

Substitute each $(s,(m_j,g_j),t)$ in Δ' s.t. $m_j \neq m_1$ and $m_j \neq m_2$ with

$(s_1,(m_j,g_j),t)$ and $(s_2,(m_j,g_j),t)$

Substitute $(s,(m_1,g_1),t_1)$ in Δ' with $(s_1,(m_1,g_1),t_1)$

Substitute $(s,(m_2,g_2),t_2)$ in Δ' with $(s_2,(m_2,g_2),t_2)$

Remove all unreachable transitions

return ($\mathcal{A}'$,null)

End

Figure 4.11. Refinement of Guarded Automata

Refine a Guarded Automaton

We present the algorithm to refine a Guarded Automaton in Fig. 4.11. The input of Refine are two transitions (with guards g_1 and g_2 respectively) which leads to the violation of autonomy on the skeleton. Refine will try to refine the current GA by splitting the source state of these two transitions. If refinement succeeds, the refined GA is returned; otherwise, a concrete error trace is returned to show that the input GA is not autonomous.

The first step of Refine is to compute the conjunction of the precondition of the two guards, i.e., $\mathsf{Pre}(g_1) \wedge \mathsf{Pre}(g_2)$. If the conjunction is satisfiable, it means that there is a possibility that at some configuration both transitions are enabled.

Then we call procedure FindPath to find a concrete error trace, which will be explained later. In the case where the conjunction is not satisfiable, we can proceed to the refinement task. We split the source state of the two transitions into two states, and modify the transitions accordingly. Finally we eliminate transitions that cannot be reached during any execution of the GA.

Example 4.29 When procedure Refine is applied to Fig. 4.9(a), and the two transitions starting at state 3, it first compute the conjunction of two preconditions: $\alpha.id \neq \beta.id \wedge \alpha.id = \beta.id$. Obviously the conjunction is not satisfiable. Then state 3 is split into two states, (state 3 and 4 in Fig. 4.9(c)), and transitions are modified accordingly. Finally, unreachable transitions (dotted arrows in Fig. 4.9(c)) are removed, and we get the GA in Fig. 4.9(c). ∎

The precondition operator Pre is a standard operator in symbolic model checking, in which, all primed variables are eliminated using existential quantifier elimination. For example given a constraint g as "$a = 1 \wedge b' = 1$", its precondition is $\mathsf{Pre}(g) = \exists_{a'} \exists_{b'} (a = 1 \wedge b' = 1)$, which is equivalent to "$a = 1$".

Generate a Concrete Error Trace

We present the algorithm to locate a concrete error trace in Fig. 4.12. Procedure FindPath has three inputs: a GA $\mathcal{A}$, a state s in $\mathcal{A}$, and a symbolic constraint g. FindPath will locate an error trace (a list of configurations) which starts from the initial state of $\mathcal{A}$, and finally reaches s in a configuration satisfying constraint g.

Procedure FindPath($\mathcal{A}$, s, g): List

Begin

Let $\mathcal{A} = (M, T, s_0, F, \Delta)$

Let $\mathcal{T}$ be $\bigcup_{(s_i,(m_j,g_k),s_\ell)\in\Delta} \quad g_k \wedge state = s_i \wedge state' = s_\ell$

Stack path = new Stack()

Let g be re-assigned as $g \wedge state = s$

$g' =$ `false`

stack.push(g)

While $g \neq g'$ and $g \ \wedge \ state = s_0$ is not satisfiable **do**

$g' = g$

$g = (\exists_{M'} \quad (g_{M/M'} \wedge \mathcal{T})) \ \vee \quad g'$

path.push(g)

End While

If $g \ \wedge \ state = s_0$ is not satisfiable **Then**

return null

Else

path = reverse order of path

List ret = new List()

cvalue = a *concrete value* of path[1]

For i =1 to $|path|$ **do**

ret.append(cvalue)

cvalue = a concrete value in $(\exists_M \quad \text{cvalue} \wedge \mathcal{T})_{M'/M}$

End For

return ret

End If

End

Figure 4.12. Generation of Concrete Error Trace

The algorithm of FindPath is a variation of the standard symbolic backward reachability analysis in model checking techniques. The procedure starts with the construction of a symbolic transition system $\mathcal{T}$, based on the control flow as well as data semantics of $\mathcal{A}$. Then given the initial constraint g, the main loop computes the constraint which generates g via transition system $\mathcal{T}$. The loop terminates when it reaches the initial configuration, or it reaches a fixed point.

Example 4.30 If we redefine the guard g_2 as $\alpha.id = \beta.id \Rightarrow \beta.id' = \alpha.id$. When procedure Refine is called on Fig. 4.9(a), the conjunction of preconditions of g_1 and g_2, i.e., $\alpha.id = \beta.id$, is satisfiable. Hence procedure FindPath is called with input Fig. 4.9(a), state 3, and constraint $\alpha.id = \beta.id \wedge state = s3$. The while loop of FindPath eventually includes in variable *path* the following constraints:

1. $\alpha.id = \beta.id \wedge state = s3$.
2. $\alpha.id = 1 \wedge state = s2$.
3. $state = s1$.

Then the order of *path* is reversed, and a concrete value constraint *cvalue* is randomly generated which satisfies constraint $state = s1$ and each message attribute has an exact value in *cvalue*, for example, let *cvalue* be $\alpha.id = 1 \wedge \beta.id = 0$, then the list *ret* will record the following constraints:

1. $\alpha.id = 1 \wedge \beta.id = 0 \wedge state = s1$.
2. $\alpha.id = 1 \wedge \beta.id = 0 \wedge state = s2$.
3. $\alpha.id = 1 \wedge \beta.id = 1 \wedge state = s3$.

Obviously the sequence of constraints in *ret* captures an error trace leading to the state 3 which violates the autonomy condition. ∎

Complexity of the algorithms in Fig. 4.10, Fig. 4.11, and Fig. 4.12 depends on the data domains associated with the input GA. When the message alphabet of a guarded conversation protocol is finite, algorithms in Fig. 4.11 are guaranteed to terminate. For infinite domains, a constant loop limit can be used to terminate Procedure FindPath by force, which will result in a conservative analysis algorithm.

4.5.2 Symbolic Analysis of Synchronous Compatibility

One natural question is: do we have similar iterative analysis algorithms for the lossless join and synchronous compatibility conditions? The answer is negative, because of Propositions 4.19, 4.18, 4.20, and 4.21. However, we do have "conservative" symbolic analyses for these two conditions. In the following, we first discuss the symbolic analysis for synchronous compatibility.

Recall the algorithm to check synchronous compatibility of an FSA conversation protocol. We project the protocol to each peer and determinize them (including ϵ-transition elimination), and then construct the Cartesian product from these deterministic projections. Then we check whether each state in the product is an illegal state (where some peer is not receptive to a message that another peer is ready to send). Note that determinization is a necessary step, otherwise the algorithm will not work.

The analysis of synchronous compatibility for a GA conversation protocol follows exactly the same procedure. But note that, we have to discuss two different cases on GA conversation protocols with finite or infinite domains. Given an F-GA conversation protocol $\mathcal{P}$, we can always construct its exact equivalent FSA conversation protocol (let it be $\mathcal{P}'$), and use the synchronous compatibility analysis for standard FSA protocols to analyze $\mathcal{P}'$. However, for I-GA conversation protocols we might not be able to do so, because there may not exist projections for I-GA conversation protocols. In the following, we introduce a "conservative" symbolic analysis for the synchronous compatible condition.

Given an I-GA (or F-GA) conversation protocol $\mathcal{P}$, we can project it to each peer using coarse projection (either Coarse Processing 1 or Coarse Processing

2 in Fig. 4.4). Then we determinize each peer in $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}1}$ (or $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}2}$) using the DeterminizeGA in Fig. 4.6. We construct the GA Cartesian product of those determinized GA using the algorithm presented in Section 4.2.1. If no illegal state is found, the I-GA conversation protocol $\mathcal{P}$ is synchronous compatible; however, this method is conservative, i.e., if an illegal state is found, $\mathcal{P}$ might still be synchronous compatible, because a coarse projection accepts a superset of the language accepted by the exact projection.

4.5.3 Symbolic Analysis of Lossless Join

Similar to the analysis of synchronous compatibility, for an F-GA conversation protocol, we can always construct its equivalent FSA conversation protocol and apply the lossless join check for FSA conversation protocols. Now we discuss a conservative and symbolic analysis for I-GA as well as F-GA conversation protocols.

Recall that each GA $\mathcal{A}$ can be regarded as a transition system, and can be represented symbolically. Let $\mathcal{T}(\mathcal{A})$ denote the symbolic transition system derived from $\mathcal{A}$. From the initial configuration of $\mathcal{A}$, we can compute all the reachable configurations of $\mathcal{T}(\mathcal{A})$, and let set of reachable configurations be $S^{\mathcal{A}}$. Given $\mathcal{A}_1$ and $\mathcal{A}_2$, the following statement is true:

$$(S^{\mathcal{A}_1} \wedge \mathcal{T}(\mathcal{A}_1) \;\Rightarrow\; S^{\mathcal{A}_2} \wedge \mathcal{T}(\mathcal{A}_2)) \;\Rightarrow\; (L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)) \tag{4.1}$$

Intuitively, Equation 4.1 means that if $\mathcal{A}_2$ as a transition system is a superset of $\mathcal{A}_1$, i.e., for any reachable configuration, there are more enabled transitions in $\mathcal{T}(\mathcal{A}_2)$ than $\mathcal{T}(\mathcal{A}_1)$, then $L(\mathcal{A}_2)$ should be a superset of $L(\mathcal{A}_1)$.

Equation 4.1 naturally implies the symbolic analysis algorithm. Given a GA conversation protocol $\mathcal{P}$ (with finite or infinite domains), let its GA specification be $\mathcal{A}$. We can project $\mathcal{A}$ using coarse projection. Then construct the Cartesian product of $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}1}$ (or $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}2}$), and let it be $\mathcal{A}'$. Then we construct $\mathcal{T}(\mathcal{A})$, $\mathcal{T}(\mathcal{A}')$, and compute $S^{\mathcal{A}}$ and $S^{\mathcal{A}'}$. Finally if $S^{\mathcal{A}'} \wedge \mathcal{T}(\mathcal{A}')) \Rightarrow (S^{\mathcal{A}} \wedge \mathcal{T}(\mathcal{A})$, we can conclude that $\mathcal{P}$ is lossless join.

The above symbolic analysis algorithm is decidable when domain is finite. When $\mathcal{P}$ has infinite domain, we can simply use the approximate closure of $S^{\mathcal{A}}$ and $S^{\mathcal{A}'}$, and it is still a conservative algorithm.

4.5.4 Hybrid Analyses

Up to now, we have introduced a range of techniques to analyze each of the realizability conditions. The most light-weight analysis is the skeleton analysis. However, this approach may not be very precise. One possibility is to use symbolic analyses to improve skeleton analysis. Given a GA-conversation protocol $\mathcal{P}$, we can first check the autonomous condition using the iterative refined analysis algorithm. If $\mathcal{P}$ is proved to be autonomous, we can apply the skeleton analysis on the refined protocol obtained in the iterative analysis. When the skeleton analysis succeeds, $\mathcal{P}$ is guaranteed to be realized by $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}1}$, $\mathcal{S}_{\mathcal{P}}^{\text{PROJ},\mathbb{C}2}$, and $\mathcal{S}_{\mathcal{P}}^{\text{PROJ}}$.

4.6 Synchronizability Analysis

The synchronizability analysis for GA web service compositions derives from the realizability analysis for GA conversation protocols. However, interestingly,

the results obtained here are better than those for conversation protocols. For example, skeleton analysis for bottom-up specified GA web service compositions does not require additional conditions (in contrast to the deterministic guards condition required in realizability analysis). For another example, the symbolic analysis for the three realizability conditions only gives the information on "whether the GA conversation protocol is realizable or not", it does not provide a synthesis solution (and for I-GA conversation protocols there may not even exist exact projections). In the bottom-up framework, we do not have an additional synthesis step – once the GA web service composition is proved to be synchronizable, we can directly do the LTL verification using the synchronous semantics (e.g. set the channel size to 0 in SPIN [41, 43]). In the following, we briefly redefine the synchronizability conditions, and present the technical details.

Definition 4.13 *Let* $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ *be a GA web service composition.* $\mathcal{S}$ *is said to be* synchronizable *if it produces the same set of conversations under both the asynchronous and the synchronous communication semantics, i.e.,* $\mathcal{C}(\mathcal{S}) = \text{JOIN}(L(\mathcal{A}_1), ..., L(\mathcal{A}_n))$.

The following definitions of synchronizability conditions are taken from Chapter 3 and modified accordingly for the GA framework.

1) Synchronous compatible condition: A GA web service composition $\langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ is synchronous compatible if for each $i \in [1..n]$, each word $w \in \Sigma^*$, and each message $\alpha \in \Sigma_a^{\text{out}} \cap \Sigma_b^{\text{in}}$:

$$(\forall i \in [1..n]\ \pi_i(w) \in L(\mathcal{A}_i^*)) \ \wedge\ \pi_a(w\alpha) \in L(\mathcal{A}_a^*) \ \Rightarrow\ \pi_b(w\alpha) \in L(\mathcal{A}_b^*),$$

where $\mathcal{A}_i^*$ is the prefix automaton of $\mathcal{A}_i$.

2) Autonomous condition: Let $\mathcal{P} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ be a GA composition. $\mathcal{P}$ is autonomous if for each peer $\mathcal{A}_i$, and for each word $w \in \Sigma_i^*$, exactly one of the following three statements holds: (a) w is accepted by $\mathcal{A}_i$. (b) there exists $\beta \in \Sigma_i^{\text{in}}$ s.t. $w\beta \in L(\mathcal{A}_i^*)$. (c) there exists $\alpha \in \Sigma_i^{\text{out}}$ s.t. $w\beta \in L(\mathcal{A}_i^*)$.

A GA web service composition is synchronizable if the above two conditions are satisfied, as formalized in the following theorem. Note that, compared with Theorem 4.11, Theorem 4.31 works for both finite and infinite domains.

Theorem 4.31 A GA web service composition is synchronizable if it satisfies the autonomous and synchronous compatible conditions.

Proof: The proof (by contradiction) follows exactly the same argument as that of Theorem 3.12. Note that even if the message alphabet Σ is infinite, it does not affect the correctness of the argument. ∎

Next we discuss the skeleton analysis for GA web service compositions. Given a GA web service composition $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, its *skeleton composition*, written as skeleton($\mathcal{S}$) is an FSA composition $\langle (P, M), \mathcal{A}'_1, \ldots, \mathcal{A}'_n \rangle$ where for each $i \in [1..n]$, $\mathcal{A}'_i$ is generated by dropping guards from $\mathcal{A}_i$.

Theorem 4.32 A GA web service composition is synchronizable if its skeleton satisfies the autonomous condition and synchronous compatible conditions[2].

Proof: The theorem is proved using contradiction, following exactly the same argument as the proof (paragraph 2) of Theorem 4.23. Basically we can show that if there is a message not consumed eagerly, we can always find a corresponding bad run in the skeleton($\mathcal{S}$). ∎

[2]Note that, the synchronizability of the skeleton composition, however, does not imply the synchronizability of the GA composition.

Symbolic analysis of each of the synchronizability conditions can follow the algorithms given in Section 4.5. Different than the symbolic analysis for realizability conditions (which does not have the synthesis algorithm for I-GA conversation protocols), once the synchronizability conditions are proved, we can construct the Cartesian product of all peers (using the algorithm in Section 4.2.1), and verify LTL properties on the Cartesian product.

Chapter 5

Expressive Power of Guarded Automata Composition

This chapter studies a variation of the Guarded Automata model proposed in Chapter 4. In the new GA model, each Guarded Automaton is allowed to have a finite set of local variables, and for simplicity, each message class is abstract. From the perspective of optimization, one natural question is: given a GA web service composition where each peer is implemented using n variables, can the composition be optimized using $n-1$ variables? We provide some initial results that are obtained in [44]. We show that whether a GA composition can be optimized is decided by the arithmetic constraints used in transition guards, and it is affected by the composition schema of the GA composition. We begin this chapter by setting the context for the presentation of technical results. Then the following two sections discuss the influence of arithmetic system and composition schema on the expressive power hierarchy of GA web service compositions.

5.1 Guarded Automata with Local Variables

We consider a variation of the GA composition called V-GA composition, where "V" stands for the "local variables". Formally, a V-GA composition is defined as follows.

Definition 5.1 *A V-GA web service composition is a tuple* $\langle (P, M), \mathcal{A}_1, ..., \mathcal{A}_n \rangle$, *where* (P, M) *is a standard composition schema, and for each* $i \in [1..n]$, $\mathcal{A}_i$ *is a* Guarded Automaton with Local Variables (called "V-GA"). *Each* $\mathcal{A}_i$ *can be represented by a tuple* $(M_i, T_i, s_i, F_i, \vec{V}_i, \delta_i)$ *where* M_i, T_i, s_i, F_i, $\vec{V}_i$, *and* δ_i *are the set of message classes, set of states, initial state, set of final states, set of local variables, and transition relation respectively. A transition in* δ_i *is one of the three types:* $(t, (!m, g), t')$, $(t, (?m, g), t')$, $(t, (\epsilon, g), t')$ *where* $t, t' \in T$, $m \in M$, *and* g *is a predicate in the form of*

$$g(\vec{V}', \vec{V}),$$

where $\vec{V}'$ *and* $\vec{V}$ *are the "next value" and "current value" of the local variables respectively.*

Note that in Definition 5.1, message classes are abstract. This chapter, in particular, study the following two types of local variables:

1. *Counter* variables: an *assignment* for a counter can only increment or decrement the value by 1; an *atomic formula* can test if a variable has the value 0.
2. *Integer* variables: an *assignment* can assign to an integer variable an arithmetic expression with $+, -, \times, \div, \sqrt{}$; an *atomic formula* can compare two

arithmetic expressions using $\leq, =$. (Note that $x \div y = k$ such that $x = yk + r$ for some $0 \leq r < |y|$ and $\sqrt{x}$ is defined if $x \geq 0$ and $= k$ where k is the largest integer such that $k^2 \leq x$.)

We use V-GA$_i^{\mathbb{C}}$ (and V-GA$_i^{\mathbb{Z}}$) to denote the family of V-GA with at most i local counter (integer) variables. Each automaton in V-GA$_i^{\mathbb{C}}$ (V-GA$_i^{\mathbb{Z}}$) is called a *counter (integer) V-GA*. Given a V-GA composition $\mathcal{S} = \langle (P, M), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, we say $\mathcal{S} \in$ V-GA$_i^{\mathbb{C}}$ ($\mathcal{S} \in$ V-GA$_i^{\mathbb{Z}}$) if for each $i \in [1..n]$, $\mathcal{A}_i \in$ V-GA$_i^{\mathbb{C}}$ (resp. $\mathcal{A}_i \in$ V-GA$_i^{\mathbb{Z}}$). For each $i \in \mathbb{N}$ and each composition schema S we denote by $\mathcal{C}_i^{\mathbb{C}}(S)$ (or $\mathcal{C}_i^{\mathbb{Z}}(S)$) the family of conversation sets $\{\mathcal{C}(\mathcal{S}) \mid \mathcal{S} \in$ V-GA$_i^{\mathbb{C}}$ (resp. $\mathcal{S} \in$ V-GA$_i^{\mathbb{Z}}$) is a V-GA composition over composition schema $S\}$.

We can have a less restricted form of V-GA$^{\mathbb{C}}$ called augmented counter V-GA. An *augmented counter V-GA* is identical to a counter V-GA except that assignments in a transition can increment or decrement a counter variable by c for some constant $c \in \mathbb{Z}$ and atomic formulas can test if a counter equals some constant c'. An augmented V-GA may use a finite number of constants (different constants for different assignments). The following lemma shows the equivalence between the two versions of counter V-GA.

Lemma 5.1 Each augmented V-GA with i counters is equivalent to a V-GA with i counters.

Proof: Let $\mathcal{A}$ be an augmented V-GA with i counters. We can effectively convert $\mathcal{A}$ to a V-GA $\mathcal{A}'$ using the following idea. For simplicity, we assume the counter values are positive (negative values can be remembered by finite states) and the increments and decrements are positive. Let c be the largest constant used in $\mathcal{A}$.

For each counter v_i in $\mathcal{A}$, $\mathcal{A}'$ has a corresponding counter v_i' to record the value of v_i modulo c, and $\mathcal{A}'$ encodes the remainder of dividing v_i by c into its states. Clearly, for each transition in $\mathcal{A}$ which increments (decrements) the value of v_i by a certain constant c_0, we have a corresponding transition in $\mathcal{A}'$ to increment (decrement) the value of v_i' by at most 1, and move to a corresponding state which encodes the corresponding remainder. ∎

Augmented V-GA can be further relaxed into an equivalent V-GA automaton model with bounded ϵ-transitions, as shown in the following lemma.

Lemma 5.2 Let $\mathcal{A}$ be a V-GA with i counters and ϵ-moves. Then the language recognized by $\mathcal{A}$ with at most d consecutive ϵ-moves can be recognized by some $\mathcal{A}'$ in V-GA$_i^{\mathbb{C}}$.

Proof: The basic idea of constructing $\mathcal{A}'$ from $\mathcal{A}$ is as follows. We examine the transitions of $\mathcal{A}$, and for every possible sequence X of transitions of at most d ϵ-moves, we can determine (actually pre-calculate by hand) the "net effect" of X on each counter and the terminal state. Note that the effect on each counter can be at most an increment of d or decrement of d. Hence, the sequence X can be replaced by one step (which gets rid of the ϵ-moves), and we get an augmented V-GA $\mathcal{A}''$. By Lemma 5.1, $\mathcal{A}''$ can be converted to an equivalent $\mathcal{A}'$ in V-GA$_i^{\mathbb{C}}$. ∎

5.2 Variable Based Hierarchies

This section focuses on the hierarchies of $\mathcal{C}_i^{\mathbb{C}}(S)$ and $\mathcal{C}_i^{\mathbb{Z}}(S)$. We consider integer and counter cases separately. For the integer case, we show that $\mathcal{C}_i^{\mathbb{Z}}(S)$ collapses at the level 1. For the counter case, using a result of Greibach [48], we show that

$\mathcal{C}_i^{\mathbb{C}}(S)$ is strict for schema S that contain one peer sending two message classes to another peer.

5.2.1 Hierarchy of $\mathcal{C}_i^{\mathbb{Z}}$

With pairing functions, a Guarded Automaton with one integer variable can simulate any Guarded Automaton with an arbitrary number of integer variables.

Lemma 5.3 For each $i \in \mathbb{N}$, every $\mathcal{A}$ in V-GA$_i^{\mathbb{Z}}$ is equivalent to some $\mathcal{A}'$ in V-GA$_1^{\mathbb{Z}}$, i.e., $L(\mathcal{A}) = L(\mathcal{A}')$.

Proof: Let $\mathcal{A} = (M, T, s, F, \vec{V}, \Delta)$ in V-GA$_i^{\mathbb{Z}}$ and $\vec{V} = \{v_1, ..., v_k\}$. We construct $\mathcal{A}' = (M, T, s, F, \{v\}, \Delta')$ which simulates $\mathcal{A}$ on every move. Consider the following pairing function $f(x, y)$ defined in [58], where f is a polynomial and x, y can be recovered by expressions using $+, -, \times, \div, \sqrt{\ }$.

$$f(x,y) = ((x+1) \div 2 + y)^2 + x, \quad f_x(z) = z - (\sqrt{z})^2, \quad f_y(z) = \sqrt{z} - (f_x(z)+1) \div 2$$

By consecutive applications of the above pairing functions, we can easily encode $v_1, \ldots, v_k$ into v, and each v_i ($i \in [1..k]$) can be decoded from v. Then, the guards in each transition of $\mathcal{A}'$ can be constructed accordingly. ∎

Theorem 5.4 For each nontrivial schema S with a peer prototype having at least two output message classes, $\mathcal{C}_0^{\mathbb{Z}}(S) \subsetneqq \mathcal{C}_1^{\mathbb{Z}}(S) = \cup_{i \geq 0} \mathcal{C}_i^{\mathbb{Z}}(S)$.

Proof: The collapse part of the proof follows from Lemma 5.3. The inequality follows from the fact that each V-GA in V-GA$_0^{\mathbb{Z}}$ generates a regular language. For S, let p_i be the peer which has at least two output message classes, and let

them be m_1 and m_2 respectively. Consider the family of languages on alphabet $\{m_1, m_2\}$, its intersection with $\mathcal{C}_0^{\mathbb{Z}}(S)$ contains regular languages only, because p_i has no counter. However, its intersection with $\mathcal{C}_1^{\mathbb{Z}}(S)$ contains a conversation set $\{{m_1}^n {m_2}^n \mid n \geq 0\}$, which can be easily generated by counting the number of m_1 and m_2 in p_i, and making all other peers to receive passively. ∎

Remark: It appears that $\sqrt{}$ is needed for the encoding to work. It is unclear if for restricted versions, the hierarchy collapses. One case is to allow expressions to be built from $+, -, \times$, and $\div$ only. Another case is to further restrict to $+, -$ (Presburger arithmetic). While these problems remain open, it is easy to construct, for each V-GA in V-GA$_i^{\mathbb{C}}$, an equivalent V-GA in V-GA$_1^{\mathbb{Z}}$ that uses only $+, -$, multiplication and division by constants for its variables. ∎

5.2.2 Hierarchy of $\mathcal{C}_i^{\mathbb{C}}$

We now turn to counter V-GA in the remainder of the section. We consider a special family of composition schema where a single peer is the sender of all messages, and other peers receive passively. We show that, for such composition schema, the hierarchy of $\mathcal{C}_i^{\mathbb{C}}$ is strict.

Languages generated/accepted by counter V-GA are closely related to languages accepted by counter machines in real time. Fischer, Meyer, and Rosenberg [34] studied real-time deterministic counter machines and showed that there is a strict hierarchy based on the number of counters. In particular, the languages used to separate each level use a two-symbol alphabet. This result on deterministic counter machines does not carry over to the nondeterministic case for counter V-GA. However, a result by Greibach [48] can be used to establish

the non-collapsing result (Theorem 5.7).

Let $\Sigma = \{a, b, c\}$. Consider the following languages from [48]:

$$\mathcal{L}_{2\ell+1} = \{wcw^R \mid w \in \{a,b\}^*\} \cap (a^+b^+)^\ell a^+c\ \{a,b\}^*, \text{ and}$$
$$\mathcal{L}_{2\ell+2} = \{wcw^R \mid w \in \{a,b\}^*\} \cap (a^+b^+)^{\ell+1} c\ \{a,b\}^*.$$

In the rest of the chapter, we call this language family the *Greibach language family*, and each language in this family is denoted as $\mathcal{L}_i$ where $i \geq 0$.

Lemma 5.5 [48] $\mathcal{L}_i = L(\mathcal{A})$ for some $\mathcal{A} \in \text{V-GA}_i^{\mathbb{C}}$, but for every $\mathcal{A}' \in \text{V-GA}_{i-1}^{\mathbb{C}}$, $\mathcal{L}_i \neq L(\mathcal{A}')$.

Note that the Greibach language family uses an alphabet of size 3. We can further improve the result to an alphabet of size 2 by modifying the languages $\mathcal{L}_i$ in the following manner: Let $\mathcal{L}'_{2\ell+1}$ be $\mathcal{L}_{2\ell+1}$ with c replaced by bb, and $\mathcal{L}'_{2\ell+2}$ be $\mathcal{L}_{2\ell+2}$ with c replaced by aa. The resulting languages $\mathcal{L}'_i$ are over alphabet $\{a, b\}$.

Next we prove that the hierarchy of $\mathcal{C}_i^{\mathbb{C}}$ is strict, based on the language family $\mathcal{L}'_i$ (which is over an alphabet of size 2).

Lemma 5.6 $\mathcal{L}_j = L(\mathcal{A})$ for some $\mathcal{A} \in \text{V-GA}_i^{\mathbb{C}}$ iff $\mathcal{L}'_j = L(\mathcal{A}')$ for some $\mathcal{A}' \in \text{V-GA}_i^{\mathbb{C}}$.

Proof: We first argue that given some $\mathcal{A}' \in \text{V-GA}_i^{\mathbb{C}}$ such that $L(\mathcal{A}') = \mathcal{L}'_j$, we can construct an augmented V-GA $\mathcal{A}''$ with i counters such that $L(\mathcal{A}'') = \mathcal{L}_j$. The inverse direction is proved similarly.

Consider the case when j is even. The idea is for $\mathcal{A}''$ to simulate the computation of $\mathcal{A}'$ faithfully while also keeping track of the number of (a^+b^+) segments that have been generated (by encoding states). When $j/2$ segments have been

generated, $\mathcal{A}''$ generates c and simulates the updating of the counters (by $\mathcal{A}'$ on aa) in one step. Thus, $\mathcal{A}''$ may have to increment/decrement some counters by 2 and test some counters also against 1 or -1 (depending on whether the corresponding transition in $\mathcal{A}'$ is to increment or decrement the counters). By Lemma 5.1, we can convert $\mathcal{A}''$ to a standard counter V-GA. Therefore there is an $\mathcal{A} \in \text{V-GA}_i^{\mathbb{C}}$ such that $L(\mathcal{A}) = \mathcal{L}_j$. ∎

Theorem 5.7 If S is a schema containing a peer sending two message classes to another peer, then for each $i \in \mathbb{N}$, $\mathcal{C}_i^{\mathbb{C}}(S) \subsetneqq \mathcal{C}_{i+1}^{\mathbb{C}}(S)$.

Proof: Let p_1 and p_2 be the two peers in S where p_1 sends messages a and b to p_2. We now prove that $\mathcal{L}'_{2i+1}$ belongs to $\mathcal{C}_{i+1}^{\mathbb{C}}$. We let p_1 have $i+1$ counters to remember the lengths of $(i+1)$ a-strings, and let p_2 to have i counters to remember the lengths of b strings. Clearly, the implementation of these two peers can guarantee that the conversation set of their composition is $\mathcal{L}'_{2i+1}$.

We now show that $\mathcal{L}'_{2i+1} \notin \mathcal{C}_i^{\mathbb{C}}(S)$. Suppose there is a counter V-GA composition $\mathcal{S}' \in \text{V-GA}_i^{\mathbb{C}}$ such that $\mathcal{C}(\mathcal{S}') = \mathcal{L}'_{2i+1}$. Then in $\mathcal{S}'$, p_1, p_2 have a total of at most $2i$ counters. Since p_1 only sends and p_2 only receives, their Cartesian product (let it be $\mathcal{A}'$) can recognize their conversation set (i.e., $\mathcal{S}'$ is synchronizable). Obviously, $\mathcal{A}'$ has at most $2i$ counters, and $L(\mathcal{A}') = \mathcal{L}'_{2i+1}$. This is a contradiction to Lemmas 5.5 and 5.6. ∎

5.2.3 Closure Properties of $\mathcal{C}_i^{\mathbb{C}}$

We now investigate the closure properties of $\mathcal{C}_i^{\mathbb{C}}(S)$. Interestingly, the closure properties provide another perspective to understand Theorem 5.7. We say that

A *homomorphism* h from $\Sigma \rightarrow \Sigma'^*$ is *nonerasing* if $h(a) \neq \epsilon$ for all in Σ.

Theorem 5.8 The language class $L(\text{V-GA}_i^{\mathbb{C}})$ is closed under nonerasing homomorphism and inverse nonerasing homomorphism.

Proof: Given a V-GA $\mathcal{A}$ with i counters that generates L, an a nonerasing homomorphism h, we construct a V-GA $\mathcal{A}'$ with i counters and ϵ-moves generating $h(L)$ as follows.

Let $\Sigma = \{a_1, ..., a_n\}$ be the alphabet of L. $\mathcal{A}'$ nondeterministically generates a string of the form $w = h(a_{i_1})h(a_{i_2})...h(a_{i_m})$, and for each $h(a_{i_k})$ where $1 \leq k \leq m$, the sequence of moves of $\mathcal{A}'$ simulate one move of $\mathcal{A}$ for a_{i_k}. $\mathcal{A}'$ enters an accepting state after generating w iff $\mathcal{A}$ enters an accepting state after generating $a_{i_1}....a_{i_m}$.

For inverse homomorphism, suppose $\mathcal{A}$ is a V-GA with i counters generating a language $L' \subseteq \Sigma'^+$, and h is a nonerasing homomorphism from $\Sigma \rightarrow \Sigma'^+$.

From $\mathcal{A}$, we construct $\mathcal{A}'$ with i counters and ϵ-moves as follows. $\mathcal{A}'$ will generate a string of the form $x = a_{i_1}...a_{i_m}$ by first generating a_{i_1} (the symbol a_{i_1} is chosen nondeterministically) after which, it goes through internal transitions for $|h(a_1)| - 1$ ϵ-moves (without generating a symbol) simulating the actions of $\mathcal{A}$ on the string segment $h(a_{i_1})$. $\mathcal{A}'$ follows the same procedure for each of the other symbols $a_{i_2}, ..., a_{i_m}$ it generates. By this way, $\mathcal{A}'$ accepts each word $w \in \Sigma$ where $h(w) \in L'$, i.e., $L(\mathcal{A}') = \bigcup_{h(L)=L'} L$. Clearly $\mathcal{A}'$ has bounded delay, which can then be converted to a machine in $\text{V-GA}_i^{\mathbb{C}}$. ∎

Definition 5.2 *Let h be a nonerasing homomorphism $\Sigma \rightarrow \Sigma'^+$. h is a* code *if $h(a) \neq h(b)$ for every pair $a \neq b$ in Σ, and each word $x \in \{h(a) \mid a \in \Sigma\}^+$ has a unique factorization in terms of the $h(a)$'s.*

Example 5.9 Let $\Sigma = \{a_1, ..., a_n\}$ be an alphabet, where $n \geq 2$. Consider the mapping $h : \Sigma \rightarrow \{0,1\}^+$ defined by $h(a_i) = 10^i1$ for $i \in [1..n]$. It is easy to see that h is a code. Similarly, let $h'(a_i)$ be the binary representation of integer i possibly with leading 0's to make all of length $\log_2 n$. Then h' is a code. ∎

Corollary 5.10 Let h be a code from $\Sigma \rightarrow \Sigma'^+$. Then $L \subseteq \Sigma^+$ can be generated by a V-GA with i counters if and only if $h(L)$ can be generated by a V-GA with i counters.

Proof: Since h is a code. It is not hard to see that, for any language $L \subseteq \Sigma^*$, $L = \bigcup_{h(L')=h(L)} L'$. Then the corollary directly follows Theorem 5.8. ∎

Corollary 5.10 directly leads to Corollary 5.11 and Corollary 5.12. Note that, Corollary 5.12 is actually the Theorem 5.7, and its proof (based on Theorem 5.8, Corollary 5.10, and Corollary 5.11) provides a general perspective of understanding the use of a smaller alphabet (than the Greibach language family) to prove the result in Theorem 5.7.

Corollary 5.11 Let h be a code from $\Sigma \rightarrow \{0,1\}^+$. Then $L \subseteq \Sigma^+$ can be generated by a V-GA with i counters but not by a V-GA with $(i-1)$ counters if and only if $h(L) \subseteq \{0,1\}^+$ can be generated by a V-GA with i counters but not by a V-GA with $(i-1)$ counters.

Corollary 5.12 If S is a schema containing a peer sending two message classes to another peer, then for each $i \in \mathbb{N}$, $\mathcal{C}_i^{\mathbb{c}}(S) \subsetneqq \mathcal{C}_{i+1}^{\mathbb{c}}(S)$.

5.3 Peer-Wise Regular Conversations

Theorem 5.7 establishes a non-collapsing hierarchy for schema that contains one peer sending two messages to another. This result relies too much on the behavior of a single peer being too complex. This is in some sense not satisfying since the focus of web service composition is to examine the interactions between component services. For this purpose, we only restrict to conversation sets whose projection to the output alphabet of each peer is a regular language.

We present two fundamental results in this section. (1) If we only consider conversations sets that are regular languages, the $\mathcal{C}_i^{\mathbb{C}}(S)$ hierarchy collapses to $\mathcal{C}_0^{\mathbb{C}}(S)$ for all schema S. Such regular conversation sets are interesting since model checking techniques are immediately available to verify properties of compositions [40, 43]. (2) We then consider conversation sets that are context-free. We show that the technique for proving the non-collapse result in Theorem 5.7 can be used to establish an infinite $\mathcal{C}_i^{\mathbb{C}}(S)$ hierarchy for schema that contains a "path" of length 2 with at least two message classes on each single directional channel of that path. The separation is tight for the case when the path forms a loop and not tight in general.

Definition 5.3 *Let S be a schema. A language L over the message alphabet of S is* peer-wise regular *if $\pi_{M_i^{out}}(L)$ is regular for each peer p_i in S.*

For each schema S, let PR-$\mathcal{C}_i^{\mathbb{C}}(S)$ be the set of languages in $\mathcal{C}_i^{\mathbb{C}}(S)$ that are peer-wise regular. The question is whether PR-$\mathcal{C}_i^{\mathbb{C}}(S)$ forms an infinite hierarchy. We start with the following example.

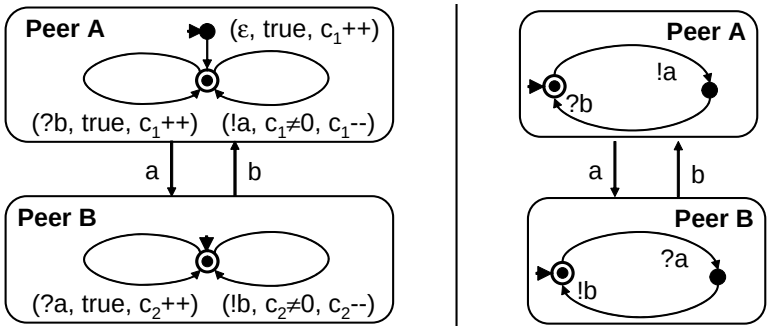


Figure 5.1. A V-GA$_1^{\mathbb{C}}$ Composition and Its Simplified Version

Example 5.13 A composition involving two peers is shown in Fig. 5.1 (the left side), where each peer has one counter to control the number of message a and b resp. For convenience, we use an ϵ-move for peer A. It is not hard to see that the language "accepted" (i.e., ignore the actions of send and receive but keep the message class) by peer A is the set $\{(a|b)^* \mid \text{in each prefix } |a| \leq |b|+1\}$, and peer B "accepts" the language $\{(a|b)^* \mid \text{in each prefix } |b| \leq |a|\}$. The conversation set generated by the two peers is however a regular set $(ab)^*$. In this case, both peers can be simplified to eliminate the counter. At the right side of Fig. 5.1 the simplified version of the two peers is given. It can be verified that they generate the same conversation set as the original peers. ∎

5.3.1 Regular Conversation Set Case

An interesting question arising from Example 5.13 is whether in a given V-GA composition, peers can be simplified with fewer counters. We now consider this problem in the case when the conversation set is known to be regular.

Theorem 5.14 Given any schema S and V-GA composition $\mathcal{S}$ over S, if $\mathcal{C}(\mathcal{S})$ is regular, there exists a counter-free V-GA composition $\mathcal{S}' \in$ V-GA$_0^{\mathbb{C}}$ over S such

that $\mathcal{C}(\mathcal{S}) = \mathcal{C}(\mathcal{S}')$.

Proof: The theorem directly follows from Theorem 3.9. Since $\mathcal{C}(\mathcal{S})$ is realizable, it is realized by its projected composition, which is counter-free. ∎

5.3.2 Context-free Conversation Set Case

We now consider the case where the conversation sets are context-free. We show that in general, it is not possible to reduce the number of counters to some fixed constant.

We start with two special schema. The first schema $S_{3\text{p}}$ consists of three peers: p_1, p_2, and p_3 where p_1 sends messages a, b to p_2, p_2 sends c, d to p_3, and p_3 only receives messages. The second schema $S_{2\text{p}}$ has two peers p_1 and p_2, sending messages to each other. In $S_{2\text{p}}$, $M_1^{out} = M_2^{in} = \{a, b\}$, and $M_2^{out} = M_1^{in} = \{c, d\}$. The following can be shown for these two schema.

Lemma 5.15 For the schema $S_{3\text{p}}$ ($S_{2\text{p}}$) and each $i > 0$, there exists a context-free language L over $\{a, b, c, d\}$ such that the following hold:

1. $\pi_{\{a,b\}}(L)$ and $\pi_{\{c,d\}}(L)$ are regular,
2. $L = \mathcal{C}(\mathcal{S})$ for some $I \in \text{V-GA}^{\mathbb{C}}_{3k+1}$ of $S_{3\text{p}}$ (resp. $\text{V-GA}^{\mathbb{C}}_{k+1}$ of $S_{2\text{p}}$), and
3. $L \neq \mathcal{C}(\mathcal{S})$ for each $\mathcal{S} \in \text{V-GA}^{\mathbb{C}}_{k}(\mathcal{S})$ of $S_{3\text{p}}$ (resp. $\text{V-GA}^{\mathbb{C}}_{k}$ of $S_{2\text{p}}$).

Proof: We first prove the $S_{3\text{p}}$ case. Consider language

$$L = \{a^{i_1}ba^{i_2}b\cdots a^{i_{3k+1}}b\ c^{i_{3k+1}}d\cdots c^{i_2}dc^{i_1}d \mid \forall \ell \in [1..(3k+1)], i_\ell \in \mathbb{N}, i_\ell > 0\}.$$

The language L is modified from the example used in the proof of Theorem 5.7. Note that item (1) is obvious. L requires that peer p_2 remembers the length

of every a^* sequence (separated by message b) sent by p_1, and sends to p_3 with similar sequences of c^*. Item (2) can be easily shown. By a reasoning similar to that in [48], L cannot be accepted in real-time by a counter machine with $\leq 3k$ counters.

We now prove by contradiction that L cannot be generated by peers with k variables, i.e., item (3). Assume that there is a k-counter implementation $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$ for peer p_1,p_2 and p_3 resp. It is not hard to see that during the interaction $\mathcal{A}_2$ cannot send out messages until all a and b messages have been sent from $\mathcal{A}_1$; otherwise they can generate a conversation which is not contained in L because c appears in the $(a^+b^+)^{3k+1}$ part. This implies that the input queue of $\mathcal{A}_1$ ($\mathcal{A}_2$) is empty whenever $\mathcal{A}_1$ ($\mathcal{A}_2$) sends out messages. Based on the results of [16], the Cartesian product of $\mathcal{A}_1$, $\mathcal{A}_2$ and $\mathcal{A}_3$, which is now a Guarded Automaton with $3k$ variables, accepts in real time the conversation set generated by the asynchronous composition of the three peers, i.e., L. This contradicts with the fact that L cannot be accepted by any counter machine with $3k$ or less variables in real time, and hence concludes the proof.

For schema $S_{2\mathrm{p}}$, let

$$L = \{a^{i_1}b^{j_1}\cdots a^{i_{k+1}}b^{j_{k+1}}\ c^{j_{k+1}}d^{i_{k+1}}\cdots c^{j_1}d^{i_1} \mid \forall \ell \in [1..k{+}1], i_\ell > 0, j_\ell > 0\}$$

and let the implementation of p_1 and p_2 each remember half of the necessary counters, then by a similar argument as the proof of Theorem 5.7, we can show a strict hierarchy of $\mathcal{C}_k^{\mathbb{C}} \subsetneq \mathcal{C}_{k+1}^{\mathbb{C}}$. ∎

Lemma 5.15 immediately leads to the following.

Theorem 5.16 For each schema S containing a path of length 2 in which the number of message classes is at least 2 for each single directional peer to peer

channel, the PR-$\mathcal{C}_i^{\mathbb{C}}(S)$ hierarchy does not collapse for context-free conversation sets. In particular, if S contains $S_{2\mathrm{p}}$ (modulo isomorphism), the hierarchy is strict at each level.

This chapter presents an initial study on comparing behavior models. Our preliminary results imply that (1) if the global behaviors of a web service composition are regular, there is no need to have local variables in the published behavior specification, but (2) if the global behaviors are context free, there is a strict hierarchy based on the number of variables used when encoding of the variables are not allowed. Results such as these are clearly important in formulating behavior models especially for the web service standards. Our study leads to a number of interesting questions. For examples, can the gap in Theorem 5.16 be narrowed? In general, how to characterize the effect of queue? In addition, there are also a number of associated complexity problems.

Chapter 6

Modeling of XML Data Manipulation

It is generally agreed that messages exchanged among web based systems should be in the XML [26] format. For example, almost all web service standards (e.g. WSDL [80], BPEL4WS [12], WSCI [79], OWL-S [24]) are built on XML and related standards including XML Schema [82] and XPath [81]. The rich tree-structured data representation of XML and powerful XPath expressions, however, impede direct application of model checking techniques to the verification of Web based systems. Earlier efforts to verify web services (e.g. [36, 65, 55]) basically focus on only the control flows by abstracting away the XML data semantics during analysis.

To model the XML data manipulation semantics of web services, this chapter, based on [40] and [41], extends the Guarded Automata model introduced in Chapter 4 in the following two ways: (1) the contents of a message class can be

organized using a complex type declared using XML Schema, and (2) a Guarded Automaton can have a finite set of local XML variables when it is used to specify a peer. This extended GA model, called the *XML-GA* model, is very expressive and allows transformation from most popular web service specification standards. It is the core part of the Web Service Analysis Tool (WSAT). In WSAT, input web services will be translated into XML-GA before any analysis or verification can be applied.

The chapter is organized as follows. We will first discuss the formal modeling of XML, bounded XML Schema, and a fragment of XPath query language. Then we propose the XML-GA model, give the syntax of XML-GA input of WSAT, and present a sample XML-GA conversation protocol. Finally, to illustrate the expressiveness of the model, we introduce the translation algorithm from static BPEL4WS web services to XML-GA.

6.1 Modeling of XML Related Standards

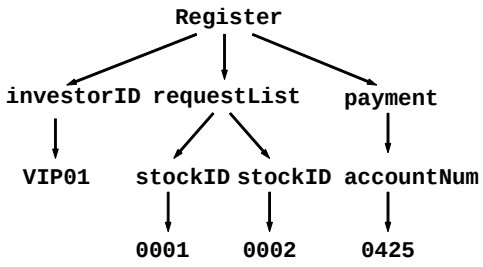
This section introduces the formal model for XML, MSL (an theoretical model of XML Schema), and a fragment of XPath.

6.1.1 XML

Similar to HTML, all XML documents are structured using tags, which are written as `<tag>` followed by `</tag>`. However, tags in XML describe the content of the data rather than the appearance. Fig. 6.1(a) shows an XML document containing the data for a `Register` message sent from an investor to register for

```
<Register>
  <investorID>
    VIP01
  </investorID>
  <requestList>
    <stockID>
      0001
    </stockID>
    <stockID>
      0002
    </stockID>
  </requestList>
  <payment>
    <accountNum>
      0425
    </accountNum>
  </payment>
</Register>
```

(a)



(b)

$l = \{$ Register, investorID, VIP01, requestList, stockID, 0001, stockID, 0002, payment, accountNum, 0425 $\}$
$n = 11$

	1	2	3	4	5	6	7	8	9	10	11
p	0	1	2	1	4	5	4	7	1	9	10
r	11	3	3	8	6	6	8	8	11	11	11

(c)

Figure 6.1. An XML document (a), the corresponding tree (b), and its formal representation (c)

a stock analysis service. It consists of a string containing the identification of the investor, a list of stock identifiers that the investor is interested in, and a record of payment information.

XML documents can be modeled as trees where each internal node corresponds to a tag and leaf nodes correspond to basic type values. The document in Fig. 6.1(a) corresponds to the tree in Fig. 6.1(b). We now present a formal representation for XML documents. Since a tag attribute can be regarded as a leaf child node of the corresponding tag node, we omit tag attributes in the model. This simplification does not weaken the expressive power.

Definition 6.1 *An XML document is a quadruple* $\mathcal{X} = (l, n, \mathrm{p}, \mathrm{r})$ *where*

1. l *is a list of* labels *where each label can either be an internal node tag, or a leaf node value with a basic type (such as boolean, integer or string). We denote* i'*th node of* l *with* $l[i]$ *(indices start from* 1*).*

2. n *is the size of* l*.*

3. *parent function* $\mathrm{p} : [1, n] \rightarrow [0, n-1]$ *is a function such that*

   (a) $\mathrm{p}(1) = 0$*, and*

   (b) *for each* $1 < i \leq n$*,* $1 \leq \mathrm{p}(i) < i$

   We define p^* *as the transitive and reflexive closure of* p.

4. *range function* $\mathrm{r} : [1, \mathrm{n}] \rightarrow [1, \mathrm{n}]$ *is a function where*

   (a) $\mathrm{r(i)} \geq \mathrm{i}$ *for each* $i \in [1, n]$*, and*

   (b) *for each* $i \leq j \leq \mathrm{r(i)}$*,* $i \in \mathrm{p}^*(j)$*, and for each* $j \notin [i, \mathrm{r(i)}]$*,* $i \notin \mathrm{p}^*(j)$*.*

Given a node at index i, p(i) points to its parent node. Since root has no parent, we define p(1) = 0. Given node i, r(i) denotes the maximum index of the nodes in the subtree of node i. Note that constraints on p and r guarantee that l is the *pre-order* traversal of the document tree of $\mathcal{X}$.

Example 6.1 Fig. 6.1(c) is the quadruple representation of the XML document in Fig. 6.1(a). Obviously the list l is the pre-order traversal of the tree in Fig. 6.1(b), and parent function p and range function r describe the tree structure. For example, the subtree starting from node `requestList` spans over five nodes, and hence the range function r(4) = 8. ∎

Definition 6.1 can be extended to describe a *tree sequence*, when restriction "$1 \leq$ p(i)" in 3(b) is modified to "$0 \leq$ p(i)". In a tree sequence, we call each node whose parent node is 0 a *root node*. Next, we introduce a *split* operator which splits a tree sequence into two tree sequences, and an *extract* operator which extracts the contents (a tree sequence) from a single XML tree.

Definition 6.2 *Given an XML tree sequence* $\mathcal{X} = (l, n, \mathrm{p}, \mathrm{r})$*, a* split *at integer* s *can be applied to* $\mathcal{X}$ *if node* s *is a root node and* $s \neq 1$*. The results are two tree sequences* $\mathcal{X}_1 = (l_1, n_1, \mathrm{p}_1, \mathrm{r}_1)$ *and* $\mathcal{X}_2 = (l_2, n_2, \mathrm{p}_2, \mathrm{r}_2)$ *where*

1. $l_1 = l[1, s{-}1]$ *and* $l_2 = l[s, n]$.
2. $n_1 = s{-}1$*, and* $n_2 = n{-}s{+}1$.
3. p_1 *coincides with* p *on domain* $[1, s{-}1]$*, and for each* $i \in [1, n{-}s{+}1]$*:*

$$\mathrm{p}_2(i) = \begin{cases} \mathrm{p}(i{+}s{-}1){-}s{+}1 & \text{if p(i+s}{-}1){\neq}0 \\ 0 & \text{if p(i+s}{-}1){=}0 \end{cases}$$

4. r_1 *coincides with* r *on domain* $[1, s{-}1]$, *and* $\mathrm{r}_2(\mathrm{i}) = \mathrm{r}(\mathrm{i}{+}\mathrm{s}{-}1){-}\mathrm{s}{+}1$ *for all* i *in* $[1, n{-}s{+}1]$.

Given an XML tree sequence which has at least m root nodes, for any $k \leq m$ we can split the tree sequence into k sequences, by consecutively applying the split operator $k-1$ times to the second part of the result of the previous split. We call this operation a k-split.

Definition 6.3 *Given a single XML tree* $\mathcal{X} = (l, n, \mathrm{p}, \mathrm{r})$, *the* extract *operator generates a tree sequence* $\mathrm{extract}(\mathcal{X}) = (l', n', \mathrm{p}', \mathrm{r}')$ *where* $l' = l[2, n]$, $n' = n{-}1$, *and for each* $i \in [1, n'] : \mathrm{r}'(\mathrm{i}) = \mathrm{r}(\mathrm{i}{+}1){-}1$ *and* $\mathrm{p}'(i) = \mathrm{p}(i{+}1){-}1$.

Example 6.2 If we apply the exact operator to the XML tree in part (c) of Fig. 6.1, we get the XML tree sequence $\mathcal{X}' = (l', n', \mathrm{p}', \mathrm{r}')$ where

$l' =$ { `investorID, VIP01, requestList, stockID,`
`0001, stockID, 0002, payment, accountNum, 0425` }

$n' = 10$

	1	2	3	4	5	6	7	8	9	10
p'	0	1	0	3	4	3	6	0	8	9
r'	2	2	7	5	5	7	7	10	10	10

Note that, the tree sequence $\mathcal{X}'$ can be split at 3 and 8, and there exists a 3-split for $\mathcal{X}'$. ∎

6.1.2 XML Schema and MSL

XML provides a standard way to exchange data over the Internet. However, the parties that exchange XML documents still have to agree on the *type* of the

data, i.e., what are the tags that will appear in the document, in what order, etc. XML Schema [82] is a language for defining XML data types. Model Schema Language (MSL) [14] is a compact formal model that captures most features of XML Schema. We use a slightly simplified version of MSL with type expressions defined as follows:

$$g \;\rightarrow\; b \;\mid\; t\texttt{[}g_0\texttt{]} \;\mid\; g_1\{m,n\} \;\mid\; g_1\texttt{,}\ldots\texttt{,}g_k \;\mid\; g_1\texttt{|}\ldots\texttt{|}g_k$$

Here $g, g_0, g_1, \ldots, g_k$ represent MSL types, b is a basic data type such as string, integer or boolean, t is a tag, and m and n are two positive integers where $m \leq n$. Intuitively, the semantics of the above MSL type syntax rules can be summarized as follows: $t\texttt{[}g_0\texttt{]}$ denotes a type with a root node labeled with t and children with types that match the sequence of MSL types represented by g_0; $g_1\texttt{\{}m,n\texttt{\}}$ denotes a sequence of size at least m and at most n where each member is of type g_1; $g_1\texttt{,}\ldots\texttt{,}g_k$ denotes an ordered sequence where the first member is of type g_1, the second member is of type g_2, and so on; and, $g_1\texttt{|}\ldots\texttt{|}g_k$ denotes a choice among types g_1 to g_k. To simplify our presentation, we will assume that the types $g_1, \ldots, g_k$ are derived by the rules "$g \rightarrow b$" or "$g \rightarrow t\texttt{[}g_0\texttt{]}$".

Similar to XML, we can define a "parent function" for MSL types. Given two MSL types g and g_i, $\mathrm{p}(g_i) = g$ if there exists a g' such that either of the following two conditions are satisfied: 1) $g \rightarrow t[g'] \;\wedge\; g' \rightarrow g_1\texttt{,}\ldots\texttt{,}g_i\texttt{,}\ldots\texttt{,}g_k$, or 2) $g \rightarrow t[g'] \;\wedge\; g' \rightarrow g_1\texttt{|}\ldots\texttt{|}g_i\texttt{|}\ldots\texttt{|}g_k$. We associate an attribute called "*tag*" with each MSL type g. If g is derived from syntax $g \rightarrow t[g_0]$, $g.tag = t$; otherwise *tag* is set `null`.

Formally, an XML document tree sequence $\mathcal{X} = (l, n, \mathrm{p}, \mathrm{r})$ is an *instance* of an MSL type g if one of the following holds:

1. when $g \rightarrow b$: $n = 1$ and $l[1]$ is a leaf node value and its type is b.

2. when $g \rightarrow t\texttt{[}g_0\texttt{]}$: $\mathcal{X}$ is a single XML document tree where $l[1] = t$ and extract($\mathcal{X}$) is an instance of g_0.

3. when $g \rightarrow g_1\texttt{\{}m, n\texttt{\}}$: there exists a k-split on $\mathcal{X}$ for some integer $m \leq k \leq n$ such that the resulting tree sequences $\mathcal{X}_1, \ldots, \mathcal{X}_k$ are all instances of g_1.

4. when $g \rightarrow g_1\texttt{,} \ldots \texttt{,} g_k$: there exists a k-split of $\mathcal{X}$, such that the resulting tree sequences $\mathcal{X}_1, \ldots, \mathcal{X}_k$ are instances of $g_1, \ldots, g_k$ respectively.

5. when $g \rightarrow g_1\texttt{|} \ldots \texttt{|} g_k$: $\mathcal{X}$ is an instance of g_i for some integer $i \in [1, k]$.

Example 6.3 It is easy to verify that the XML document `Register` presented in Fig. 6.1 is an instance of the following MSL type.

```
Register[
 investorID[string],
 requestList[ stockID[int]{1,3} ],
 payment[ creditCard[int] | accountNum[int] ]
]
```

6.1.3 XPath

In order to write specifications or programs that manipulate XML documents we need an expression language to access values and nodes in XML documents. We use a subset of XPath [81] to navigate through XML trees and return the answer nodes. The fragment of XPath we use consists of the following operators:

the child axis (`/`), the descendant axis (`//`), self-reference (`.`), parent-reference (`..`), basic type test (`b()`), node name test (t), wildcard (`*`), and predicates `[]`. An *XPath expression* is defined with the following grammar

$$
\begin{aligned}
exp &\rightarrow p \mid exp\ \texttt{op}\ exp \\
p &\rightarrow r \mid \texttt{/}r \mid \texttt{//}r \\
r &\rightarrow s \mid r\texttt{/}s \mid r\texttt{//}s \\
s &\rightarrow \texttt{.} \mid \texttt{..} \mid n^{?}(\texttt{[}exp\texttt{]})^{*} \mid \texttt{position()} \mid \texttt{last()} \\
n &\rightarrow b\texttt{()} \mid t \mid *
\end{aligned}
$$

where $n^{?}$ denotes n or empty string and $(\texttt{[}exp\texttt{]})^{*}$ denotes zero or more repetition of `[`exp`]`.

In the above syntax rules, *exp* denotes an *XPath arithmetic expression* which is constructed by combining *XPath location paths* (represented by p) with arithmetic operators (represented by `op`). There are two types of location paths: relative location paths and absolute location paths. An absolute location path starts with `/` or `//`. A relative location path (represented by r) consists of a list of steps (represented by s) which are connected with `/` or `//`. The steps in a relative location path are evaluated from left to right. A step can be a self-reference (`.`), a parent-reference (`..`), or a more complex form which consists of a node test (n) and a sequence of predicates of the form [*exp*]. A node test n has three possible forms: type test (b`()`), name test (t), and wildcard match ($*$). Finally, a step can be a function call such as `position()` or `last()` with the following restriction: Function calls can only appear as the last step of a location path.

Formally, an XPath expression accepts inputs of the form (c, d) where *context* c is a *sequence* of node indices in some XML document $\mathcal{X}$, and d is either a single node in $\mathcal{X}$ or a set of values with the same basic type. The sequence of node indices in the input context must be in ascending order with no repetition. The output of an XPath expression is a sequence of nodes in the same XML document used in the input, or a set of values. For $\mathcal{X}(l, n, \mathrm{p}, \mathrm{r})$, let N be the domain of all node indices in $\mathcal{X}$ (i.e., $[1, n]$), and DOM be the domain of all leaf node values. Then the semantics of an XPath expression *exp* (as well as a step, a node test, and a location path) can be defined as a function:

$$exp : 2^N \times (N \cup 2^{DOM}) \ \rightarrow \ 2^N \cup 2^{DOM}.$$

Before we formally define the semantics of XPath expressions, we will give some example expressions and the results of evaluating them below. To distinguish node indices from other values we write node indices in bold characters.

Example 6.4 Given input $(\{\mathbf{1}\}, \mathbf{1})$, where **1** is the root node of the XML document presented in Fig. 6.1, and the following XPath expressions

1. `investorID`
2. `//stockID[position() = 2]/int()`
3. `//stockID/int() = 0002`

The results are {**2**}, {0002}, and {`false, true`} respectively. ∎

We will now present the formal semantics of XPath expressions bottom-up. We will present the semantics of a node test first, and then a step, and then a location path, and finally an XPath expression.

Given a node test n and input (c, d) where d is a node of XML document $\mathcal{X}(l, n, \mathrm{p}, \mathrm{r})$, the results of $n(c, d)$ is defined as follows:

1. when $n \rightarrow$ `b()`: $n(c, d) = \{d' \mid \mathrm{p}(d') = d \ \wedge \ \ l[d']$ is of type `b`$\}$.
2. when $n \rightarrow$ `t`: $n(c, d) = \{d' \mid \mathrm{p}(d') = d \ \wedge \ l[d'] = t\}$.
3. when $n \rightarrow$ `*`: $n(c, d) = \{d' \mid \mathrm{p}(d') = d\}$

Basically, rule 1 and 2 select the children nodes of the input node d by their types and tags (resp.), and rule 3 simply returns all the children nodes. Finally, when input d is a set of values, then $n(c, d)$ returns the empty set $\emptyset$.

The definition of a step s is similar. Given a step s and input (c, d), when d is a set of values, s returns $\emptyset$. When d is a node of $\mathcal{X}$, $s(c, d)$ is defined as below:

1. when $s \rightarrow$. : $s(c, d) = \{d\}$.
2. when $s \rightarrow$ .. : $s(c, d) = \{\mathrm{p}(d)\}$.
3. when $s \rightarrow [exp]$:
$$s(c,d) = \begin{cases} \{d\} & \texttt{true} \in exp(c,d) \text{ and } exp \text{ is a boolean expression} \\ \{d\} & exp(c,d) \neq \phi \text{ and } exp \text{ is a locationpath} \\ \emptyset & otherwise. \end{cases}$$
4. when $s \rightarrow n\ [exp_1] \ldots [exp_k]$: $s(c, d) = p'(c, d)$ where
$$p' = n \ \texttt{/}\ [exp_1] \ \texttt{/}\ [exp_2] \ \texttt{/}\ \ldots \ \texttt{/}\ [exp_k]$$
5. when $s \rightarrow$ `last()`: $s(c, d) = \{|c|\}$
6. when $s \rightarrow$ `position()`: $s(c, d) = \{$position of d in c$\}$

In rules 1 and 2, the handling of self reference and parent reference is straightforward. For the third rule, the step either returns the singleton set $\{d\}$ or an empty set, depending on the evaluation of the predicate. Note that when a location path is used as a predicate, it evaluates to `true` if it returns a non empty set of nodes. Finally the evaluation of a step which consists of node test plus a series of predicates is reduced to that of an equivalent location path. For example, the step `stockID[position()=2]` is equivalent to a location path `stockID/[position()=2]`. Given input (c, d), function call `last()` simply returns the singleton set which contains the size of context c; `position()` returns the position of d in c where position is counted starting from 1.

Given a relative path $r \rightarrow r_1/s$, and an input (c, d),

$$r(c, d) = \cup_{d' \in r_1(c,d)} \; s(r_1(c, d), d'). \tag{6.1}$$

According to the above formula the steps of a relative location path are executed one by one from left to right. Note that the context of each step is the result of the previous step. For the case where $r \rightarrow r_1//s$, we replace $r_1(c, d)$ in Equation 6.1 with the following set $\{n \mid \mathrm{p}^*(n) = n' \text{ for some } n' \in r_1(c, d)\}$. For example, given the XML document in Fig. 6.1 and the input $(\{\mathbf{4}\}, \mathbf{4})$, the first step and the descendants operator of the location path `stockID//[position()==3]` produces the new context $\{\mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}\}$, and the second predicate step generates the result $\{0001\}$.

An absolute location path is defined based on a relative location path. For an absolute location path $p \rightarrow /r$, and input (c, d), $p(c, d) = r(\{\mathbf{1}\}, \mathbf{1})$ where $\mathbf{1}$ is the root. When $p \rightarrow //r$, $p(c, d) = p'(\{\mathbf{1}\}, \mathbf{1})$ where $p' = .//r$.

Finally the semantics of an XPath arithmetic expression $exp \rightarrow exp_1$ `op` exp_2

is defined as follows:

$$exp(c,d) = \{v | v = v_1 \texttt{ op } v_2 \wedge v_1 \in exp_1(c,d) \wedge v_2 \in exp_2(c,d)\}.$$

Basically, it computes the results of all possible combinations from the value sets of the two operands exp_1 and exp_2. Note that, when used as a condition, a boolean XPath expression evaluates to `true` if its result set contains at least one `true` value[1].

6.2 The XML-GA Model

We now extend the GA model to incorporate XML data manipulation semantics. In an *XML-Guarded Automaton* (XML-GA), each message class has its type declared using MSL. In addition, an XML-GA that is used as a peer implementation can have local XML variables. Formally, the XML-GA framework is given in the following three definitions.

Definition 6.4 *An* XML-GA composition schema *is a tuple* (P, M, Σ) *where* P *is a finite set of peers,* M *is a finite set of message classes, and* Σ *is a (finite or infinite) set of messages. Each message class* $c \in M$ *has an MSL type, and each message* $m \in \Sigma$ *is an instance of some message class in* M*. Let* $\textsc{dom}(c)$ *denote the domain of message class* c*, the message alphabet* Σ *is defined as:*

$$\Sigma = \bigcup_{c \in M} \{c\} \times \textsc{dom}(c).$$

Definition 6.5 *An* XML-GA conversation protocol *is a tuple* $\langle (P, M, \Sigma), \mathcal{A} \rangle$*, where* (P, M, Σ) *is an XML-GA composition schema, and* $\mathcal{A} = (M, \Sigma, T, s, F, \delta)$

[1] XPath 2.0 (working draft) has a more delicate handling for this scenario. There are two sets of arithmetic/comparison operators: one to support the XPath 1.0 semantics (presented in this paper); the other will raise a type error when any operand contains more than one value. It is not hard to support the second semantics with our approach.

is an XML-GA, where M, Σ, T, s, F, δ are the set of message classes, set of messages, set of states, initial state, set of final states, and transition relation respectively. Each transition $\tau \in \delta$ is in the form of $\tau = (s, (c, g), t)$, where $s, t \in T$ are the source and the destination states of τ, $c \in M$ is a message class and g is the guard *of the transition.*

Definition 6.6 *An* XML-GA web service composition *is a tuple $\mathcal{S} = \langle (P, M, \Sigma), \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, where (P, M, Σ) is an XML-GA composition schema, $n = |P|$, and for each $i \in [1..n]$: $\mathcal{A}_i$ is a tuple $(M_i, \Sigma_i, t_i, s_i, F_i, \vec{V}, \delta_i)$ where M_i, Σ_i, t_i, s_i, F_i, and δ_i are the set of message classes, set of messages, set of states, initial state, final states, and transition relation, respectively, and $\vec{V}$ is a set of XML local variables. Each XML local variable has an MSL type. A transition $\tau \in \delta_i$ can be one of the following three types: a send transition $(t, (!\alpha, g), t')$, a receive transition $(t, (?\beta, g), t')$, and an ϵ-transition $(t, (\epsilon, g), t')$.*

Notice that the XML-GA used in Definition 6.5 does not have local variables, because it is pointless to have local variables at the global level, considering that a conversation protocol will be used to synthesize peer implementations. Guards here are expressed in a different way than Chapter 4 – there is no primed form of message attributes, and each guard is built upon XPath expressions. In the XML-GA model, each guard g consists of two parts: a transition condition and an assignment, i.e, g can be written as $g \equiv g_1 \Rightarrow g_2$. Let p_i be the sender of the message that is being sent by the transition which g belongs to. g_1 is a boolean XPath expression on the message classes (as well as local variables if the XML-GA is used a peer) related to p_i. g_2 is an assignment of the form p:=exp where p is an XPath location path (applied on a local variable or an output

message of p_i), and *exp* is an XPath location path or an XPath expression. Such representation of guards in the XML-GA model is consistent with the syntax of XPath standards, and it caters to the WSAT input format, which is introduced as below.

6.2.1 Syntax of WSAT Input

In Fig. 6.2, we present the abstract syntax of the WSAT input language for XML-GA conversation protocols. The syntax rules for XML-GA compositions are similar.

Spec → `{` *Schema* `,` *Protocol* `}`

Schema → `Schema{ PeerList{` *StringList* `},TypeList{` *MslExpList* `},`
`MessageList{` *MessageList* `}}`

MessageList → *Message* | *Message* `,` *MessageList*

Message → *name* `{` *source* `->` *destination* : *type* `}`

Protocol → `Protocol{ States{` *StringList* `},InitalState{` *StringList* `},`
`FinalStates{` *StringList* `},TransitionRelation{` *TransitionList* `}}`

TransitionList → *Transition* | *Transition* `,` *TransitionList*

Transition → *name* `{` *source* `->` *destination* : *message* `,` *Guard* `}`

Guard → `Guard{` *XPathExp* `=>` *Update* `}`

Update → *name* `{` *AssignList* `}`

AssignList → *Assign* | *Assign* `,` *AssignList*

Assign → *XPathExp* `:=` *XPathExp*

Figure 6.2. Syntax of XML-GA Conversation Protocols

Clearly, a conversation protocol specification of WSAT consists of a compo-

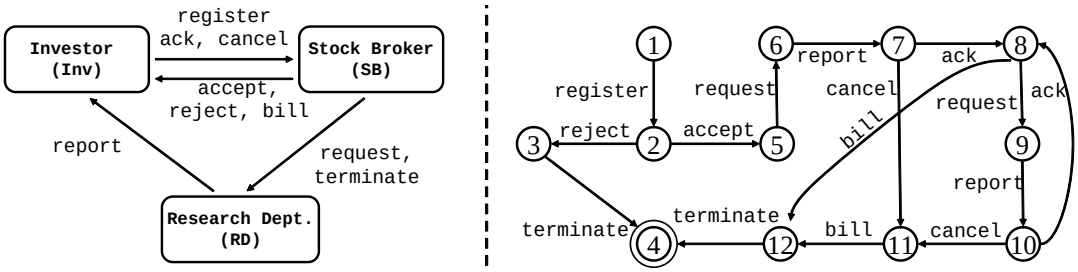


Figure 6.3. Stock Analysis Service

sition schema that specifies the peers and the message classes exchanged among peers, and the protocol (XML-GA) which specifies the desired message exchange sequences. In Fig. 6.2, The nonterminals *name*, *source*, *destination*, *type*, and *message* all denote strings. The nonterminal *StringList* denotes a list of strings separated by commas, and the nonterminal *MslExpList* denotes a list of MSL expressions separated by commas. The nonterminal *XPathExp* denotes an XPath expression. In a valid specification, *source* and *destination* should be state names, *type* should be name of an MSL type and *message* should be a message name all defined in the specification. A better explanation of the syntax rules in Fig. 6.2 is the Stock Analysis Service example that is presented in the following.

6.2.2 Stock Analysis Service – A Case Study

We now present an XML-GA conversation protocol: Stock Analysis Service (SAS). Fig. 6.3 presents its overall structure and control flow, and Fig. 6.4 is a fragment of its formal specification. As shown in Fig. 6.3, SAS involves three peers: Investor (Inv), Stock Broker Firm (SB), and Research Department (RD). Inv initiates the stock analysis service by sending a `register` message to SB.

```
Conversation {
 Schema{
  PeerList{Inv,SB,RD},
  TypeList{
   Register[
    investorID[xsd:string],
    requestList[
     stockID[xsd:int]{1,3}
   ],
   payment [
     accountNum[xsd:int] |
     creditCard[xsd:int]
   ],
  },
  ...
  MessageList{
   register{Inv -> SB: Register},
   reject{SB -> Inv: Reject},
   ...
  }
 },

 Protocol{
  States{s1,s2,...,s12},
  InitialState {s1},
  FinalStates{s4},
  TransitionRelation{
   ...
  t8{s8 -> s9 : request,
    Guard{
     $request//stockID/int() !=
       $register//stockID [position() = last()]/int() =>
     $request[
      //investorID := $register//investorID,
      //stockID :=
       $register // stockID
         [ position() = $register // stockID
             [int()=$request//stockID/int()]/position()+1
         ]
    }
  },
  t14{ s8 -> s12 : bill,
     Guard{
      $request//stockID =
         $register//stockID [position() = last()] =>
      $bill[
            //orderID:= $register//orderID
          ]
     }
  },
   ...
  }
 }
}
```

Figure 6.4. A Fragment of SAS Specification

SB may `accept` or `reject` the registration. If the registration is accepted, SB sends an analysis `request` to RD. RD sends the results of the analysis directly to Inv as a `report`. After receiving a `report`, Inv can either send an `ack` to SB or `cancel` the service. Then, SB either sends the `bill` for the services to Inv, or continues the service with another analysis `request`. In Fig. 6.4 we present a partial specification of the SAS protocol. The specification of SAS consists of two parts: a schema and an XML-GA protocol. The schema specifies the set of peers, a list of MSL types, and a list of peer to peer message classes which are built upon the MSL types. The XML-GA specification consists of states, and transitions. We present two key transitions from the protocol: `t8` and `t14`. A transition is equipped with a guard which determines the transition condition and the assignment of the message being sent. For example, transition `t8` sends a message of type `request`. Its transition condition is the following boolean XPath expression:

```
$request//stockID/int() !=

   $register//stockID [position() = last()]/int()
```

The rest of `t8` assigns values to the `investorID` and the `stockID` fields of the `request` message being sent. According to the semantics of XML-GA, except the `request` message which appears at the left side of assignment operator "`:=`", the appearance of all other message classes refers to the *latest* copy of that message class. Hence the transition condition of `t8` means "if the `stockID` of the latest `request` message is not the last `stockID` of `register` message". Its assignment tries to send the `stockID` which is subsequent (in the `register` message) to the `stockID` appeared in the latest `request` message. Similarly the guard of transition `t14` specifies that if the latest `request` message contains the last `stockID` in the `register` message, then a `bill` message is sent to conclude the interaction. Generally, `t8` and `t14` intends to send `stockID`'s in `register` one by one.

6.3 From BPEL4WS to XML-GA

One natural concern about the XML-GA model would be how powerful the XML-GA model is to capture real-world web services. In the following, we give a translation algorithm, which, given a set of BPEL4WS process specifications and the related WSDL port declarations, constructs a corresponding XML-GA composition. Translation algorithm from other web service specification languages, e.g. WSCI [79], OWS-S [24], can be developed in a similar way.

The construction of the XML-GA composition schema (P, M, Σ) is straightforward. P is constructed by taking BPEL4WS process names from BPEL specifications, M are extracted from WSDL files, and we do not have to enumerate Σ. Specifically, to construct M, for each input/output/fault parameter of an operation in each port and each service link of each BPEL4WS process, a message class is declared. For example if a BPEL4WS process `loanProcess` has an operation `approve` declared in a port `aprvPT`, and its input parameter is of WSDL message format `creditInfo`, a message class `loanProcess_aprvPT_approve_In` will be declared in the composition schema, and `creditInfo` is used as its domain type. When the name of an operation is unique among ports, our tool will omit the port name in the generated message name for simplicity (e.g. the `approve_Out` in Fig. 6.5). In BPEL, the type of the contents of a message can be defined using WSDL constructs, SOAP definition or XML Schema, we translate all possible formats to MSL [14].

Next we discuss the translation of BPEL4WS control flow and data manipulation. In Fig. 6.5 we present the XML-GA translation for some typical language constructs in BPEL4WS language.

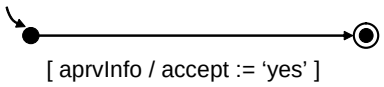
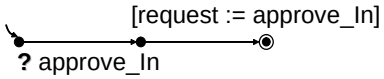
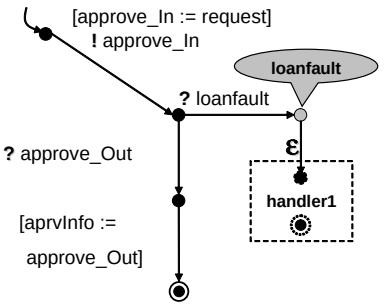
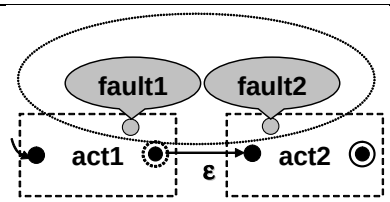
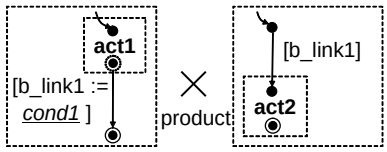
BPEL	Sample Code	Translation
assign	<assign ...> <copy> <from="yes"/> <to var="aprvInfo" part="accept"/> </copy> </assign>	
receive	<receive ... operation="approve" variable="request" />	
invoke	<scope> <invoke ... operation="approve" invar="request" outvar="aprvInfo /> <catch ... faultname="loanfault" < ... handler1 ... /> </catch> </scope>	
sequence	<sequence ... > < ... act1 ... > < ... act2 ... > </sequence>	
flow	<flow ...> < ... act1 ...> <source linkname="link1" condition="cond1"/> </act1> < ... act2 ...> <target linkname="link1/> </act2> </flow>	

Figure 6.5. From BPEL4WS to XML-GA

As shown in Fig. 6.5, each BPEL4WS language construct is translated into an XML-GA with one single initial state and one single final state. For example, the `assign` statement is translated to a one-transition automaton where the XPath expression guard of the transition expresses the assignment. Note that BPEL4WS has several different approaches for navigating messages (e.g. the keyword `part` used in the example or using XPath expressions). We translate all of them to equivalent XPath expressions, and these XPath expressions are then embedded into the guards of the generated transitions. The `receive` statement is translated into a two-transition automaton, where the first transition receives the message and the second transition assigns the input variable. Similarly, the main body of the `invoke` statement is translated to an automaton where the first transition sends the input message for the operation that is being invoked, and the following two transitions receive the response and assign the output variable (assuming there are no exceptions). Note that, exceptions might arise during `invoke`, and we have to generate additional transitions to handle them. For each fault there is a transition which leads to an "exception exit", where the information about the fault is associated with the exception exit. When a fault handler is wrapped around an `invoke` statement, our translator connects the fault handler with the corresponding exception exit.

BPEL4WS control flow constructs such as `sequence`, `switch`, and `while` are used to compose atomic constructs we discussed above. In Fig. 6.5 we display the translation for `sequence`. We connect the final state and initial state with local transitions, and unmark the final state of all activities except the last one. The information about exception exits are recollected and properly maintained. Other control flow constructs can be handled similarly by embedding the control

flow to the transitions of the XML-GA. Finally, for `flow` construct (which is the concurrent composition of its branches), we simply construct the Cartesian product of all its branches. There might be control dependency links among the activities in different flow branches. We map each link into a boolean variable, and their semantics are reflected in the guards of the transitions appended to each activity.

Translation of the control flow of BPEL4WS to finite state machines or petri-nets has been discussed in [36, 65]. The difference in our work is that we handle XML based data manipulation using XML-GA with guards expressed as XPath expressions. This enables us to verify properties about XML data manipulation. Such analysis cannot be done using approaches presented in [36, 65], since they abstract away the data content.

Chapter 7

Handle XML Data in Verification

The use of XML as the de facto data exchange standard has allowed integration of heterogeneous web based software systems regardless of implementation platforms and programming languages. On the other hand, the rich tree-structured data representation, and the expressive XML query languages (such as XPath) make formal specification and verification of software systems that manipulate XML data a challenge. This chapter presents our initial efforts [41] in formal specification and verification of software systems with XPath based manipulation of (bounded) XML data. The techniques presented in this chapter constitute the basis of our Web Service Analysis Tool (WSAT) [43, 78] which can verify Linear Temporal Logic (LTL) properties of conversation protocols [39] and interacting BPEL4WS [12] web services [40]. Clearly, these techniques can also be used for verification of other types of software systems that exchange XML data.

We use SPIN [50] as a back-end model checker in verification of XML data ma-

nipulation operations. We developed algorithms for translating XML data types and XPath expressions to Promela, the input language of SPIN. Our handling of XML data manipulation consists of two parts: (1) a mapping from XML Schema to the type system of Promela, and (2) a translation algorithm which generates Promela code for an XPath expression. The type mapping is straightforward; however, the translation of XPath expressions is not trivial. We implemented the translation algorithms presented in this chapter as a part of WSAT.

Our use of SPIN as the back-end model checker is based on the following considerations: (1) Promela supports arrays which is very useful in translating XML Schema data types. (2) The communication channels in Promela enables us to model the asynchronous communication among web services [40]. However, SPIN is an explicit-state model checker, and may not scale to large data domains due to state-space explosion. In the future we plan to investigate the use of symbolic model checking techniques in verification of XML data manipulation.

In [65], verification and composition of web services are investigated using a Petri Net model. In [36], web service compositions are specified using message sequence charts, modeled using finite state machines and analyzed using the LTSA model checker. These earlier efforts on verification of Web based software systems mostly concentrate on analysis of the control flows. Our techniques for handling XML data, however, enable verification of properties relating to data manipulation. This enables us to analyze Web based software systems at a greater level of detail without ad-hoc data abstractions. The idea of employing back-end model checkers for verification of an expressive language is used in other verification tools such as Bandera [19].

JWIG project extends the Java language with high-level features for web service programming such as dynamic construction of XML documents [21]. To ensure that the generated XML document is consistent with the message format (declared using DSD [62], a type system similar to XML Schema), JWIG provides static analysis for a set of pre-defined properties. The verification problem considered in this chapter (and in WSAT) is rather different: we consider the relationships (expressed in temporal logic) between *multiple* XML messages during the execution of a web service. Also, we focus on XPath expressions which are not part of JWIG.

The techniques presented in this chapter apply to bounded XML data only, where the number of children of an XML node is always bounded. Unbounded XML Schema types, and various fragments of XPath can be captured using unranked tree automata [59, 66]. While the unranked tree automata model overcomes the problem of boundedness, the data semantics of leaf value nodes are lost in the modeling. For example, the fragment of XPath studied in [59] does not allow arithmetic constraints in qualifiers, and it only reasons about the structure of an XML document.

This chapter is organized as follows. We first introduce the mapping from MSL (a theoretical model for XML Schema) to Promela, and then the translation algorithm from XPath to Promela is presented. Next we discuss how to translate an XML-GA to a Promela process. Then we give a case study and show how our techniques help to discover subtle errors in web service designs. Finally we briefly present WSAT to conclude the chapter.

```
typedef t1_investorID{
 mtype stringvalue;
}

typedef t2_stockID{
 int intvalue;
}
typedef t3_requestList{
 t2_stockID stockID [3];
 int stockID_occ;
}
typedef t4_accountNum{
 int intvalue;
}
typedef t5_creditCard{
 int intvalue;
}
mtype {m_accountNum, m_creditCard}
typedef t6_payment{
 t4_accountNum accountNum;
 t5_creditCard creditCard;
 mtype choice;
}
typedef Register{
 t1_investorID investorID;
 t3_requestList requestList;
 t6_payment payment;
}
```

Figure 7.1. Promela translation of Example 6.3

7.1 From MSL to Promela

In this section we focus on mapping types in MSL to Promela. We present an example translation in Fig. 7.1, and the translation algorithm is given in Fig. 7.2.

Fig. 7.1 is the Promela translation of the MSL type given in Example 6.3. Clearly each MSL basic type has a straightforward mapping to Promela. For example, `int` and `boolean` are mapped to Promela type `int` and `bool` respectively. MSL type `string` is mapped to `mtype` (enumerated type) in Promela, e.g., the leaf node value of `investorID`. In the XPath translation, which will be explained later, all string constants will be collected, declared, and used as symbolic constants of `mtype`. We assume strings are used solely as constants, and we do not expect any operators to change values of these `mtype` variables.

Translation of complex MSL types is more complicated. Generally, each MSL complex type is translated into a corresponding `typedef` (record) type in Promela. For example, the `Register` type in Example 6.3 is mapped into

```
// ret[1]: type declaration for g, including intermediate types.
// ret[2]: attribute definition for the input g if g is intermediate.
function tr(g: MSL): String[2]
begin
  String ret1, ret2;
  switch case
    g → b :
      ret1 = null;
      ret2 = b + " " + b + "value";
    g → t[g0] :
      if (g is an intermediate type) then
        type = generate a unique name;
        ret1 = tr(g0)[1] + "typedef " + type + "{" + tr(g0)[2] + "}";
        ret2 = type + " " + t;
      else
        ret1 = tr(g0)[1] + "typedef " + t + "{" + tr(g0)[2] + "}";
        ret2 = null;
      end if
    g → g1{m, n} :
      ret1 = tr(g1)[1];
      ret2 = tr(g1)[2] + "["+n+"]" + "int " + g1.tag +"_occ"
    g → g1, g2, ..., gk :
      ret1 = tr(g1)[1] + tr(g2)[1] + ... + tr(gk)[1];
      ret2 = tr(g1)[2] + tr(g2)[2] + ... + tr(gk)[2];
    g → g1 | g2 | ... | gk :
      ret1 = tr(g1)[1] + tr(g2)[1] + ... + tr(gk)[1] +
             "mtype {"+ "m_"+g1.tag + ...+ "m_"+gk.tag + "}";
      ret2 = tr(g1)[2] + tr(g2)[2] + ... + tr(gk)[2] + "mtype choice";
  end switch
  return (ret1, ret2)
end
```

Figure 7.2. Translation from MSL to Promela

Promela declaration `typedef Register`, and the intermediate type `requestList` inside `Register` is translated into `typedef t3_requestList`. Prefixes such as

`t3_` are added to prevent name collisions for intermediate types. Since each intermediate MSL type is a child of its parent type, in the Promela type declaration for its parent type, it has a corresponding attribute definition. For example, the statement "`t3_requestList requestList`" defines the attribute `requestList` in `typedef Register`. When an intermediate MSL type has multiple occurrences, e.g., the `stockID` element, it is defined as an array with its max occurrence as the array size. In addition, an additional variable (e.g. `stockID_occ`) is defined in its parent type to record its actual occurrence. For the MSL types constructed using the choice operator `|`, a variable `choice` is used to record the actual type chosen in an XML instance of the MSL type (e.g., the `choice` attribute declared in `t6_payment`).

In Fig. 7.2 we present a procedure `tr`, which takes an MSL type declaration g as its input, and generates two strings as its output. The first string, i.e., `ret[1]`, contains the type declaration for g (as well as all the necessary type declarations for its intermediate types). The second output is the attribute definition for g, if g is an intermediate type. For example, when the procedure is called for intermediate type `requestList`, `ret[1]` contains the declaration of `t2_stockID` and `t3_requestList`, and `ret[2]` contains "`t3_requestList requestList;`". (We do not show the generation of separator "`;`" in Fig. 7.2, however, it can be handled easily). As shown in Fig. 7.2, the function body of `tr` processes the input MSL type declaration recursively according to the syntax rules. Note that, it properly handles the issues such as array declaration for types with multiple occurrences and complex types constructed using choice operator.

7.2 From XPath to Promela

In this section we present the translation algorithm from XPath to Promela. We start with a brief discussion of the use of XPath expressions in XML manipulating software, then we study a motivating example, and finally we present the translation algorithm.

Consider the use of XPath in languages with XML data manipulation such as BPEL4WS and WSCI. There are basically two types of usage: 1) boolean XPath expressions are used in branch or loop conditions, and 2) location paths and arithmetic expressions are used on the left and right hand sides of assignment statements, respectively. We handle these two cases separately since the semantics of XPath expressions can depend on the context they are used. For example, when a location path is used as a boolean condition its meaning is different than the case where it is used on the left hand side of an assignment. Since the implementation of these two cases are similar, in the remainder of this section, we concentrate only on the translation of boolean XPath expressions.

7.2.1 A Motivating Example

Consider the following XPath boolean expression where the XML variable `register` is of MSL type `Register` as defined in Example 6.3, and the MSL type of variable `request` consists of a single child `stockID` (in XPath the prefix `$` is used to denote variable names):

```
$request//stockID/int() =
  $register//stockID[int()>5][position()=last()]/int()        (7.1)
```

```
1  /* result of the XPath expression */
2  bool bResult = false;
3  /* results of the predicates 1, 2, and 1 resp. */
4  bool bRes1, bRes2, bRes3;
5  /* index, position(), last(), index, position() */
6  int i1, i2, i3, i4, i5;
7
8  i2=1;
9  /* pre-calculate the value of last(), store in i3 */
10 i4=0; i5=1; i3=0;
11 do
12 :: i4 < v_register.requestList.stockID_occ
13   ->
14   /* compute first predicate */
15   bRes3 = false;
16   if
17   :: v_register.requestList.stockID[i4].intvalue>5
18      -> bRes3 = true
19   :: else -> skip
20   fi;
21   if
22   :: bRes3 -> i5++; i3++;
23   :: else -> skip
24   fi;
25   i4++;
26
27 :: else -> break;
28 od;
29 /* translation of the whole expression */
30 i1=0;
31 do
32 :: i1 < v_register.requestList.stockID_occ
33    ->
34    /* first predicate */
35    bRes1 = false;
36    if
37    :: v_register.requestList.stockID[i1].intvalue>5
38       -> bRes1 = true
39    :: else -> skip
40    fi;
41    if
42    :: bRes1 ->
43       /* second predicate */
44       bRes2 = false;
45       if
46       :: (i2 == i3) -> bRes2 = true;
47       :: else -> skip
48       fi;
49       if
50       :: bRes2 ->
51         /* translation of expression */
52         if
53         :: (v_request.stockID.intvalue ==
54           v_register.requestList.stockID[i1].intvalue)
55           -> bResult = true;
56         :: else -> skip
57         fi
58       :: else -> skip
59       fi;
60       /* update position() */
61       i2++;
62    :: else -> skip
63    fi;
64    i1++;
65 :: else -> break;
66 od;
```

Figure 7.3. Promela Translation of Equation 7.1

An XPath expression following a variable name is evaluated on the value of the variable (which is an XML document) starting with the context $(\{\mathbf{1}\}, \mathbf{1})$, where $\mathbf{1}$ is the root node of the corresponding XML document. The XPath expression in Equation 7.1 queries whether in the XML document `register` the last `stockID` which has a value greater than 5 is equal to the `stockID` of `request`. Its corresponding Promela translation is shown in Fig. 7.3.

Note that we have four boolean variables and five integer variables in the Promela translation. Boolean variable `bResult` is used to record the evaluation result of the whole XPath expression, `bRes1` and `bRes2` are used for evaluation of the two predicates on the right hand side of the expression, and `bRes3` is used during the evaluation of the `last()` function. Integer variables `i1` and `i4` are used as array indices in different parts of the Promela code, `i3` records the value of function call `last()`, and `i2` and `i5` are used for `position()` function.

It is not hard to see that we can compute the value of `last()` prior to the evaluation of the whole XPath expression, and we record its value in `i3`. The main body of the calculation is a loop searching for the proper value of array index `i4` which satisfies the first predicate (value of `stockID` greater than 5).

The main body to compute the whole boolean XPath expression is similar. There is a loop searching for the proper value of array index `i1`, and the code handling two predicates are nested. Note that position variable `i2` and array index `i1` are properly updated. According to the semantics of boolean expressions in the XPath standard, `bResult` is set to true once we find a value of `i1` satisfying the boolean expression.

Finally, note that there are more efficient Promela translations than the one

presented in Fig. 7.3. For example, integer variable `i4` (for array index) can be reused to replace `i1`. In our implementation, we have a variable assignment optimizer to achieve this objective. However, we omit the details of its implementation here to simplify the presentation.

7.2.2 Supporting Data Structures

We can make the following observations based on the motivating example shown in Fig. 7.3: (1) Every XPath language construct (expression, path, step) corresponds to a Promela code segment. For example, the boolean XPath expression shown in Equation 7.1 corresponds to the whole Promela code in Fig. 7.3, its right hand side corresponds to the whole code with lines 51 to 57 left blank, and the left hand side corresponds to an "empty" statement since no code is generated for it. (2) In particular, loops are generated for those steps which generate XML data that corresponds to an MSL type with multiple occurrences (i.e., types declared as $g\{m, n\}$). For example, the step `stockID` in the right hand side corresponds to the loop from line 30 to line 66. (3) The generated code segments are embedded into each other. For example the segment (lines 34 to 63) that corresponds to "`[int()>5]`" is embedded in the code for step `stockID`; while it embeds the code for predicate "`[position()=last()]`" (lines 43 to 59). (4) The generated code can be regarded as an nested-loop which simulates the search procedures for each location path, and the evaluation of the boolean expression is placed in the body of the inner-most loop.

Our translation algorithm needs a mechanism to represent the structure of the input XML document and an approach to conveniently capture the Promela

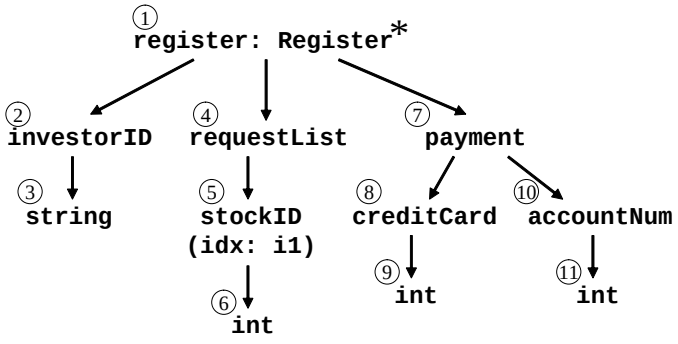


Figure 7.4. The Type-Tree for Variable `register`

code segments that are generated and embedded into each other. Hence, we introduce two data structures which will be used in the translation algorithm: a *type tree* structure which represents the MSL types and a *statement macro* which represents (partially) generated Promela code segments. We also define several functions that manipulate these data structures.

Type Tree. We use the type trees to statically represent the input and output of an XPath location path (or a step). Given an XML variable and its MSL type, it is straightforward to derive the corresponding type tree. For example, Fig. 7.4 is the corresponding type tree for XML variable `register` in Equation 7.1, with the MSL type given in Example 6.3. Note that each node in the type tree corresponds to a subexpression of the MSL type expression given in Example 6.3, where the root node corresponds to the whole type expression. Hence, each node also corresponds to an MSL type.

In a type tree, each node is labeled with an MSL type (for the root node, it is also labeled with the XML variable name). Note that if the associated MSL type has multiple occurrence, the node is equipped with an additional index. For example, index `i1` is associated with node 5 in Fig. 7.4. Recall that an MSL type

with multiple occurrence is translated into an array in Promela. This index is used to access the elements of that array. For each node in a type tree, by tracing back to the root of the tree, we can get its *qualified name*, i.e., the expression in the Promela translation which accesses the data with the type represented by that node. For example, `v_register.requestList.stockID[i1].intvalue` is the qualified name of node 6, where the prefix `v_` is automatically added by the system to avoid name collision, and the `intvalue` is the name of the attribute with the basic type `int`.

We now define a number of functions on type trees. Given a type tree t, and an XPath step s, function MarkChild(t,s) proceeds as follows: (1) unmark all marked nodes in t, and (2) for each node that is unmarked in step 1, mark its children which are the results of executing step s, and (3) return the modified t. For example, let t_r be the type tree in Fig. 7.4 where node 1 is the only marked node (marked with "*"), let s be the step "`requestList`". The result of MarkChild(t_r,s) is the same type tree where node 4 is the only marked node. Other functions such as MarkParent(t), MarkAll(t), MarkRoot(t) work in a similar way. For example, nodes 6,9, and 11 are the marked nodes after executing MarkChild(MarkAll(t_r), "`int()`").

Statement Macro. In our translation algorithm, each XPath construct corresponds to a Promela code segment. A code segment can be regarded as a list of statement macros which are sequentially concatenated using "`;`". A statement macro (or simply macro) captures a block of Promela code for a certain functionality, and each macro has at most one *BLANK* space where another Promela code segment can be embedded. There are five types of macros we are using in

our translation algorithm which are summarized below. A macro can have input parameters, and in the corresponding Promela code, the appearance of these parameters will be replaced by the actual input value when the macro is used. Fig. 7.5 presents a set of macros organized as a tree. There are two types of

Statement Macro	Promela Code

IF(v)	`if`
	`::` v `->` *BLANK*
	`::` `else -> skip`
	`fi`

FOR(v,l,h)	v `=` l `- 1`
	`do`
	`::` v `<` h `->` *BLANK*
	v `++`
	`::` `else -> break`
	`od`

EMPTY	*BLANK*
INC(v)	v `++`
INIT(v,a)	v `=` a

edges from a child to its parent: the embedment edge which is shown as a solid arrow and the sequential composition edge which is shown as a dotted arrow. We call such a tree a *macro tree*. In fact, any Promela code generated for an XPath expression construct can be captured using one or a set of macro trees. Given a macro tree, it is straightforward to generate the corresponding Promela code. For example, the macro tree in Fig. 7.5 corresponds to the code segment from line 30 to line 66 (with line 51 to line 57 as *BLANK*) in Fig. 7.3.

We associate two attributes with each macro: an input type node, and an output type node from a type tree. For each macro, the input node characterizes the starting node where the macro starts searching, and the output node is the starting point of its embedment. For example in Fig. 7.5, macro `FOR(i1,1,3)` is the corresponding code for the step "`//stockID`" in the location path at the top of the figure. Its input type node is the type node 1 in Fig. 7.4, which corresponds to the type of the XML variable `register`. Its output node (also the input node

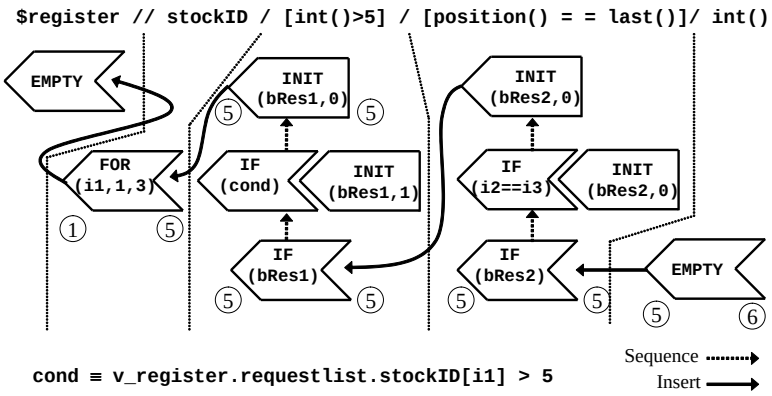


Figure 7.5. The Macro Tree for Fig. 7.3

for its embedment) is the type node 5 (with MSL type `stockID`), which is the result of evaluating the step on the input type node. In our translation, except for concatenating code for two expressions, the input node of embedded code should match the output node of the *BLANK* where it is inserted.

We also associate a hashtable with each macro, which records the mapping from XPath location paths to qualified names. For example, the hashtable of the last EMPTY macro in Fig. 7.5 will map the location path shown in Fig. 7.5 (i.e., right hand side of Equation 7.1) to qualified name of node 6, i.e., the expression "`v_register.stockID[i1].intvalue`".

We have three different functions to embed macros into each another: MatchInsert(c_1,c_2), InsertAll(c_1,c_2), and InsertAndReplace(c_1,c_2). All these functions return one macro tree that is the result of embedding c_2 into the *BLANK*s of c_1. In our translation, c_1 is always guaranteed to be a single macro tree, while c_2 can be a set of macro trees. The MatchInsert function requires that the inserted macro tree must match the output node of its host; while InsertAll and InsertAndReplace do not require matching. When inserting c_2 into the *BLANK*s

of c_1 InsertAndReplace replaces the location paths in c_2 with qualified names based on the hashtable of the host.

Function GenCode(n) generates a macro tree given a type node n. GenCode relies on a global registry R which registers index variables that are processed before. The function traces back from the current type node to the root type node. Whenever an unprocessed index is encountered, a FOR macro is generated for that index, and the index is registered in R. When a new FOR macro is generated, the old FOR macro generated before is embedded in the new macro to form a nested loop. For example, if index `i1` has not been processed, GenCode(n_6) generates a FOR(`i1`,1,3), given n_6 is the node 6 in Fig. 7.4.

7.2.3 Syntax Directed Translation Algorithm

Now we discuss the syntax directed translation algorithm which is presented in Fig. 7.6. Each non-terminal (e.g. *exp*, *p*, *s*) has one inherited-attribute: *inTree*, and two synthesized-attributes: *outTree* and *code*. Attributes *inTree* and *outTree* are both type trees, and they are used to capture the input and output of XPath language constructs, respectively. Attribute *code* is a set of macro trees, which records the generated Promela code that corresponds to the non-terminal. Non-terminal *exp* has an additional attribute *var*, which is the boolean variable that records the evaluation results for the *exp*. The *var* attribute is not null if and only if *exp* is not an intermediate expression (e.g. *exp* is used as a branch condition in host language or it is a predicate in another XPath location path). For example, when generating the Promela code (Fig. 7.3) for Equation 7.1, the attribute `var` for the boolean expressions in predicates "`[int()>5]`" and "`[position()=last()]`" are `bRes1` and `bRes2` respectively.

(1) $exp \rightarrow exp_1$ `op` exp_2:
$exp_1.inTree = exp.inTree$
$exp_2.inTree = exp.inTree$
If exp is intermediate **Then**
$exp.code$ = InsertAll($exp_1.code$,$exp_2.code$)
else
$exp.code$ =
InsertAndReplace(
InsertAll($exp_1.code$, $exp_2.code$),IF(exp)
)
where the *BLANK* of IF(exp)
is filled with "$exp.var$ `=` true"
End If

(2) $exp \rightarrow$ `op` exp_1:
$exp_1.inTree = exp.inTree$
$exp.code = exp_1.code$

(3) $exp \rightarrow$ `const` :
$exp.code$ = EMPTY

(4) $exp \rightarrow p$:
$p.inTree = exp.inTree$
$exp.code = p.code$

(5) $p \rightarrow$ `$v` p_1:
$p_1.inTree$ = generate a type tree for `$v`
$p.outTree = p_1.outTree$
$p.code = p_1.code$

(6) $p \rightarrow$ `/` p_1 | `//` p_1:
If $p \rightarrow$ `/`p_1 **Then**
$p_1.inTree$ = MarkRoot($p.inTree$)
Else
$p_1.inTree$ = MarkAll($p.inTree$)
End If
$p.code = p_1.code$
$p.outTree = p_1.outTree$

(7) $p \rightarrow p_1$ `/` s | p_1 `//` s:
$p_1.inTree = p.inTree$
If $p \rightarrow p_1$`/`s **Then**
$s.inTree$ = MarkRoot($p_1.outTree$)
Else
$s.inTree$ = MarkAll($p_1.outTree$)
End If
$p.code$ = MatchInsert($p_1.code$,$s.code$)
$p.outTree = s.outTree$

(8) $s \rightarrow$. :
$s.outTree = s.inTree$
$s.code$ = EMPTY

(9) $s \rightarrow$ .. :
$s.outTree$ = MarkParent($s.inTree$)
$s.code$ = EMPTY

(10) $s \rightarrow b() \mid t \mid *$:
$s.outTree$ = MarkChild($s.inTree$,s)
$s.code = \{c_1, \ldots, c_k\}$ where c_i is GenCode(d_i)
for each type node d_i in $s.outTree$

(11) $s \rightarrow [exp]$:
$exp.var$ = a unique variable name
$s.outTree = s.inTree$
$s.code = exp.code$ ";" IF(exp.var)

Figure 7.6. Translation from XPath to Promela

Handling of Expressions. Rules 1, 2, 3, and 4 handle the translation of XPath (boolean or arithmetic) expressions. In rule 1 both subexpressions inherit the *inTree* from *exp*. For example, when evaluating Equation 7.1 the *inTree* to inherit is null. For another example, when processing the expression "`int() > 5`", the *inTree* to inherit is a version of the type tree shown in Fig. 7.4 where the node 5 is the only marked node. We will discuss where instances of type trees are generated later in the handling of XPath location paths.

The *code* of *exp* is synthesized from the *code* of the two subexpressions. The basic idea is to embed the code generated by exp_2 into the code of exp_1, regardless of the matching of input/output type node (by the use of InsertAll instead of MatchInsert). If *exp* is not intermediate (e.g. it is used as a boolean branching condition), we need additional processing (calling InsertAndReplace) to insert another IF macro into the synthesized code. The IF macro assigns `true` to attribute *var* if the *exp* evaluates to true. Hence the generated code evaluates the boolean expression and stores the result in *var*. For example, as we mentioned earlier, the code for the right hand side of Equation 7.1 is the whole code in Fig. 7.3 except lines 51 to 57 are *BLANK*, and the code for the left hand side is an EMPTY macro. When we synthesize the *code* for Equation 7.1 from these two subexpressions, an IF macro (which assigns the *var*, i.e., the `bResult`) is embedded in that *BLANK* (lines 51 to 57) by the call to InsertAndReplace. Note that the location paths of Equation 7.1 are replaced by qualified names.

The rest of the expression related syntax rules, i.e., rules 2, 3, and 4, work in a similar way: they pass down information *inTree* to subexpressions, and synthesize *code* and *outTree* from subexpressions. Finally, note that *outTree* is not used in

the syntax rules for expressions.

Handling of Location Paths. Rules 5, 6, and 7 handle the translation of an XPath location paths. In rule 5, where a location path is associated with an XML variable, a corresponding type tree is generated and passed to the steps of the path. For example, to handle the right hand side of Equation 7.1, the type tree in Fig. 7.4 is generated. Note that even for the same XML variable, when a new type tree instance is generated, the index attributes should have unique names. For example, when pre-calculating the value of `last()` (line 11 to 28), another type tree is generated for XML variable `register`, and the index of node 5 is `i4` (instead of the `i1` in Fig. 7.4).

Rule 6 handles the absolute location paths, where the inherited attribute *inTree* is handled differently for XPath operator "`/`" and "`//`" respectively. Rule 7 processes a path, step by step and from left to right, as it passes the *outTree* of the partial path p_1 to the step s on the right. Note that when synthesizing *code*, we need to match the type node when embedding macros, so MatchInsert is called.

Handling of Steps. Rules 8, 9, 10, 11 handle steps. The semantics of rules 8 and 9 is clear. Rule 11 calls MarkChild function to symbolically execute the step s on the *inTree*. For each type node d_i in the *outTree*, function GenCode is called to generate a macro tree for d_i. Finally, rule 11 handles the case when the step is a predicate, for example, the boolean expression "`int() > 5`" (let us call it e_2) in Equation 7.1. Its synthesized code consists of two parts: an evaluation code for the expression (lines 35 to 40 for the evaluation of e_2), and an IF macro (lines 41 to 63) which allows insertion of code for later steps (lines 44 to 61).

7.2.4 Handling of Function Calls

The handling of `position()` and `last()` calls is a little bit more complicated, though the idea is similar: substitute the appearance of a function call with an integer variable, and properly update its value so that when the function is called the integer variable contains the right value.

Each `position()` (or `last()`) call has an *owner* which is an non-intermediate boolean XPath expression where the call appears. For an owner *exp*, we need another attribute called *prefix* which contains Promela macros (just like the *code* attribute). The code in *prefix* will be placed ahead of the code contained in *code* to form the complete code for the owner.

When a `position()` call is encountered, we acquire a unique integer variable for that call (let it be v). Then we append the macro INIT(v,1) in the *prefix* attribute of *owner* and insert the INC(v) in the *BLANK* of the macros generated by the immediate previous step. For example, to handle the `position()` of Equation 7.1, the integer variable `i2` is acquired, and its initialization statement is at line 8, and its update statement is at line 61 (which is inside the *BLANK* of the code that corresponds to the previous step "`[int()>5]`").

The handling of `last()` is even more complicated: it works in three modes: normal mode, copy mode, and processed mode. The normal mode is for the first time the `last()` call is encountered; in the copy mode the `last()` is encountered for a second time when the pre-calculation code is being generated; the processed mode is the case where the value for the `last()` call has been pre-calculated and this value should not be changed any more. Consider the `last()` call in Equation 7.1 as an example. In the normal mode, we acquire an integer variable for the

`last()` call (i.e., `i3`), and call the handling of its owner (i.e., Equation 7.1) to pre-calculate the value of `last()` (hence line 9 to line 28 will be generated). Now when the second translation of Equation 7.1 reaches the `last()` call, it is in the copy mode. The initialization and update statements are generated for the pre-calculation code (i.e., line 10 and line 22). When we return from the pre-calculation, the processing of the `last()` enters the processed mode, and `i3` is not allowed to be changed. The appearance of `last()` in the second predicate is replaced by `i3`.

Example 7.1 The translation of Equation 7.1 is split into two recursive translation tasks on its left and right hand paths. It is not hard to see that the left hand side generates an EMPTY macro. Now we concentrate on the right hand side, which is converted to the following form:

```
$register//stockID/[int()>5]/[position()=last()]/int()
```

The translation algorithm will start from `$register` and then processes steps from left to right. First a type tree for `register` (as shown in Fig. 7.4) is generated. Then function MarkAll is called, and the resulting tree is passed as the *outTree* to step `stockID`. In the *outTree* of step `stockID`, node 5 will be the only marked node. Then function GenCode is called for node 5, which generates a FOR macro that corresponds to lines 31 to 65 in Fig. 7.3 (with lines 34 to 63 as *BLANK*). The handling of the next step `[int()>5]` is similar, where an IF macro is generated and embedded into the FOR macro generated before. For the third step `[position()=last()]`, integer variables `i2` and `i3` are acquired for the function two calls respectively. The initialization and update statements for `position()` are generated (line 8 and line 61). However since it is in the normal mode for `last()`, we do not generate any code for `last()`. Instead,

the translation of Equation 7.1 is called again for the pre-calculation of `last()`, and lines 9 to 28 are generated. After the return from the second translation call on Equation 7.1, the first translation call advances to the last step `int` which generates an EMPTY macro whose hashtable contains the information that maps the right hand side location path to the corresponding qualified name. Finally, when synthesizing the *code* attribute for Equation 7.1, an IF macro (lines 51 to 57) is inserted and the two location paths in Equation 7.1 are replaced with qualified names. ∎

7.3 From XML-GA to Promela

Each XML-GA web service composition can be translated into a Promela specification which consists of a set of concurrent processes, one for each XML-GA. Each Promela process is associated with an asynchronous communication channel storing its input messages.

Fig. 7.7 presents the Promela translation for a "LoanProcessing" example in the BPEL4WS standard specification [12]. As explained in Chapter 5, each BPEL4WS web service is translated into an XML-GA, and each XML-GA is translated into a Promela process (i.e., the `proctype`) in Fig. 7.7. Since there are four peers (a loan approval service, a customer, a back-end approver, and a risk assessor) in the LoanProcessing example, in Fig. 7.7 there are correspondingly four process types `loanaprv`, `customer`, `approver` and `assessor`. The default main process in Promela is called `init`. The `init` process in Fig. 7.7 initializes all global variables (initialization can be non-deterministic) and spawns four processes, creating one process instance for each process type.

```
/* type declaration */
typedef creditInfo{
 mtype name; ...
}
...
/* message declaration */
creditInfo aprv_In_s, aprv_In_r, stub_aprv_In;
...
/* enumerate type of msgs and states of peers*/
mtype = {m_aprv_In, ....
         m_loanaprv_s1, ... }
mtype msg;
...
/* channels */
chan ch_loanaprv = [8] of {mtype, creditInfo, appeal};
chan ch_customer= [8] of {mtype, aprvInfo};
...
proctype loanaprv(){
 mtype state;
 /* definition of local variables */
 creditInfo request; ...
 /* definition of auxiliary variables used
    to evaluate XPath expressions */
 bool bVar_0, ...

 do::
  /* evaluation of transition conditions */
  ... bCond1 = true; ...

  /* nondeterministically select transitions to fire */
  if
   /* transition t1: s1 -> s2, ?aprv_In */
   ::state == m_loanaprv_s1 && bCond1 &&
     ch_loanaprv ? [m_aprv_In] ->
      atomic{
       ch_loanaprv ?
          m_aprv_In, aprv_In_r, stub_appeal;
       state = m_loanaprv_s2 ;
      }

   /* transition t2: s2 -> s3, !aprv_Out,
     [cond2 => aprv_Out//accept = 'yes' ] */
   ::state == m_loanaprv_s2 && bCond2 ->
     atomic{
      aprv_Out_s.accept = m_yes;
      ch_customer ! m_aprv_Out, aprv_Out_s;
      state = m_loanaprv_s3;
      msg = m_aprv_Out
     }
   ...
   /* may jump out if it is a final state */
   :: state == m_final -> break;
  fi;
 od;
}
proctype customer(){ ... }
proctype assessor(){ ... }
proctype approver(){ ... }
init{
 /* initialization */
 ...
 atomic{
  run loanaprv(); run customer(); ...
 }
}
```

Figure 7.7. An Example Promela Translation

The first part of the Promela code consists of type declarations and global variable definitions. Each MSL type declaration used in conversation schema is mapped into a record type (`typedef`) in Promela. Each message class in a conversation schema has three corresponding global variables declared: one for recording its last sent instance (e.g. `aprv_In_s` for message type `aprv_In`), one for recording its last received instance(e.g. `aprv_In_r`), and one "stub" variable used in channel operations (e.g. `stub_aprv_In`). For each message class, we also declare a corresponding enumerated constant, e.g., `m_aprv_In` for `aprv_In`. The set of all these enumerated constants constitutes the domain of enumerated variable `msg`, which is used to store the type of the latest transmitted message.

A channel variable is declared for each peer to simulate its input queue. For example channel `ch_loanaprv` is the queue of peer `loanaprv` and its length is 8. The contents of a channel includes all input message classes of that peer. In this example, peer `loanaprv` has two input message classes: `aprv_In` and `appeal`. Note that in each send/receive operation of a channel, we actually send one message only, and other elements have to be filled with stub messages. The first `mtype` element in a channel content indicates the message class that is being transmitted.

Inside each `proctype` the local variables are declared first, followed by the auxiliary variables used for the evaluation of XPath expressions. An enumerated (`mtype`) variable `state` is used to record the current state of the automaton. The main body of the process is a single loop. In each iteration of the loop, first enabling condition of each transition guard is evaluated and the result is stored in the corresponding boolean variable for that condition. For example, the `cond1` in Fig. 7.7 records the evaluation results for the enabling condition of transition

`t1`.

In Promela, `if` statements can have multiple branches with a test condition for each branch, similar to a switch statement. One of the branches of the `if` statement with a test condition that evaluates to true is nondeterministically chosen and executed. In the Promela translation for an XML-GA, each transition of the automaton is translated into a branch of the `if` statement inside the main `do` loop body. The test condition for each branch checks whether the current `state` is the source state of the corresponding transition, and whether the enabling condition of the corresponding transition evaluates to true. For receive-transitions, we check if the head of the channel contains the right message class by testing the first element of the channel content. (Note that Promela statement `channel ? messages` has side effects and cannot be used as a boolean condition, hence we have to use `channel ? [...]` statement, which checks the receive executability only but does not execute the receive operation.) If the head of the channel matches the message class of the receive operation, we consume the message, do the assignment, and update the local variable `state`. The handling of send-transitions is similar, and the only difference is that we need to update global variable `msg` while sending the message. Finally, if the state is a final state, a nondeterministic choice can be made to jump out of the loop and terminate.

Remark: When an XML-GA is used as a conversation protocol, the translation proceeds in a similar way. However, we do not have to associate channels with the corresponding Promela process, because the conversation protocol involves one automaton only. In addition, there is no local variables in an XML-GA conversation protocol.

7.4 Applications

In this section we discuss the applications of our techniques to the verification of web services. We present a case study, where our techniques help to identify a very delicate design error of XPath expressions in a conversation protocol.

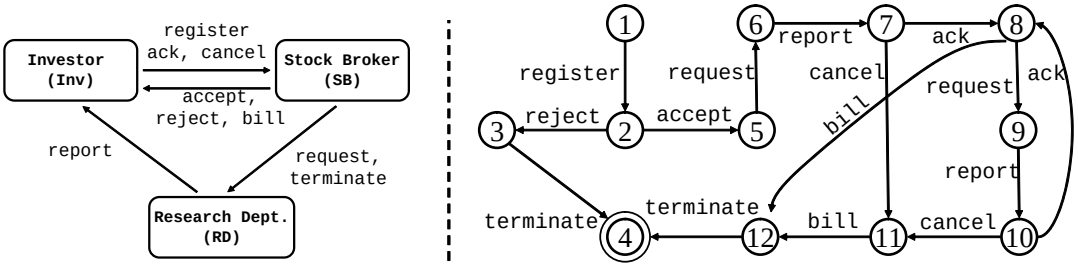


Figure 7.8. Stock Analysis Service

We pay a revisit to the SAS example discussed in Section 6.2.2. Its composition schema and control flows are replotted in Fig. 7.8. Two transitions: `t8` and `t14` are presented in Fig. 7.9.

```
t8{s8 -> s9 : request,
    Guard{
     $request//stockID/int() !=
       $register//stockID [position() = last()]/int() =>
     $request[
      //investorID := $register//investorID,
      //stockID :=
       $register // stockID
         [ position() = $register // stockID
             [int()=$request//stockID/int()]/position()+1
         ]
    }
},
t14{ s8 -> s12 : bill,
     Guard{
      $request//stockID =
         $register//stockID [position() = last()] =>
      $bill[
            //orderID:= $register//orderID
          ]
     }
}
```

Figure 7.9. Transitions `t8` and `t14`

Recall that the transition condition of `t8` means "if the `stockID` of the latest `request` message is not the last `stockID` of `register` message". Its assignment tries to send the `stockID` which is subsequent (in the `register` message) to the `stockID` appeared in the latest `request` message. Similarly the guard of transition `t14` specifies that if the latest `request` message contains the last `stockID` in the `register` message, then a `bill` message is sent to conclude the interaction. Generally, the logic of `t8` and `t14` intends to send out the list of `stockID` in the initial `register` message one by one. Given the logic of the transitions `t8` and `t14`, it is natural to propose the following LTL property for the Promela translation of the SAS protocol:

```
G (
    (
     index < v_register.requestList.stockID_occ &&
     v_register.requestList.stockID[index].intvalue == value
     && msg == m_register
    )
   ⇒
    (
     F(msg == m_reject) ||
     F(msg == m_cancel) ||
     F(request.stockID.intvalue == value)
    )
)
```

In the above LTL property, temporal operator **G** means "globally", temporal operator **F** means "eventually" and *index* and *value* are two predefined constants. The variables starting with `v_` are the qualified names referring to XML data. The variable `msg` is a variable in the Promela translation for XML-GA, which records the current message being sent. For example, when transition `t8` is executed,

`msg` will be assigned the value `m_request`.

The LTL property states that: if the `register` message contains a `stockID` (at position *index*, with value *value*), then eventually there should be a `request` containing that `stockID`, if nothing wrong happens (i.e., the `register` is not rejected, and the Inv does not cancel the service).

Interestingly, SPIN soon identifies that the SAS specification does not satisfy the proposed LTL property. SPIN gives an error-trace where the `register` message has three `stockID`s with values 0, 1, 0 respectively. The error-trace shows that when the first `request` for `stockID` 0 is sent, transition `t8` is disabled because the `stockID` of the latest `request` is the last `stockID` in the `register` message; instead, the transition `t14` is triggered to send out the `bill` message to conclude the interaction. The verification identifies the error in the design of XPath transition guards which rely on the presumption that "there should be no redundant `stockID`s in the `register` message", however this is not enforced by the specification.

As SPIN is an explicit model checker, the verification, unfortunately does not scale very well. When integer domain is set to [0,1], the verification time is 3 seconds and memory consumption is around 50MB. When the domain is increased to [0,3], the memory consumption grows to over 600MB. However, our experience shows that SPIN is still useful in identifying errors in protocols by restricting the data domains.

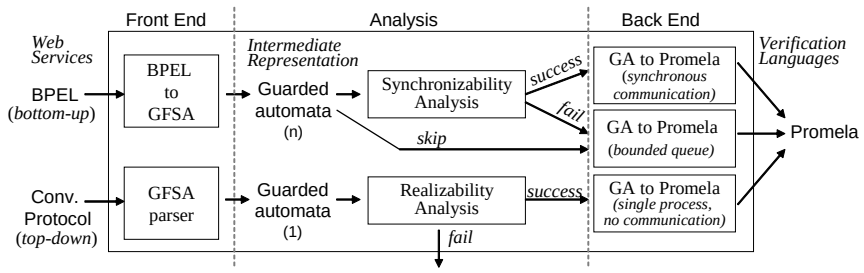


Figure 7.10. WSAT architecture

7.5 WSAT

In Fig. 7.10, we present the general architecture of WSAT. The front-end of WSAT accepts industry web service standards such as WSDL and BPEL, the core-engine of WSAT is based on the intermediate representation GA, and the back-end employs model checker SPIN as the back-end model checker. At the front-end, translation algorithm is developed from BPEL4WS to XML-GA, and support for other languages can be added without changing the analysis and the verification modules of the tool. At the core-engine part, synchronizability and realizability analyses are developed to rule-out the undecidability caused by asynchronous communication. At the back-end, translation algorithms are developed from XML-GA to Promela, the input language of SPIN. LTL verification can be performed using the synchronous communication semantics instead of asynchronous communication semantics.

In addition to the SAS example, we applied WSAT to a range of examples, including six conversation protocols converted from the IBM Conversation Support Project [51], five BPEL4WS services from BPEL4WS standard and Collaxa.com, and the SAS from [41]. Synchronizability and realizability analysis are applied to each example, and except two conversation protocols, all examples pass these checks. This implies that the sufficient conditions in our synchronizability and re-

alizability analysis are not restrictive and they are able to capture most practical applications. For each example, we generated the corresponding Promela specification using WSAT, and we checked LTL properties of the form "$\mathbf{G}(p \rightarrow \mathbf{F}q)$" using SPIN. Our experience with these examples suggests that while exhaustive search of the state space may be very costly for verifying correct properties, SPIN's performance at discovering false LTL properties is satisfactory.

Chapter 8

Conclusions

This dissertation studies the formal modeling of interacting web services, and develops a range of analyses and verification approaches which help designers to ensure the implementation of web services will meet preset mission-critical service properties.

We start from a simple automata-theoretic model, where each individual web service is modeled as a finite state automaton, and global behaviors of a web service composition are characterized by the set of conversations (i.e., sequence of send-events of messages). We have many interesting theoretical observations on this simple model, including the context-sensitive conversation set generated by an arbitrary FSA web service composition, the undecidability of LTL model checking, and the closure properties of conversation sets. In contrast to the conventional bottom-up specification approach, we propose the notion of a conversation protocol to specify desired set of message exchange sequences. A conversation protocol, as a weaker specification approach[1], has certain benefits in the analysis

[1]A conversation protocol is "weaker" because each realizable conversation protocol has a

of web services. The simple automata-theoretic model has many variations and extensions (including the Büchi, F-GA, I-GA, V-GA, and XML-GA web service compositions), which either address different features or model web services at a different detail level.

To avoid the undecidability that is caused by the asynchronous communication, we develop two analyses (on FSA and Büchi models) for the bottom-up and top-down specification approaches, respectively. We show that when a set of sufficient synchronizability conditions are satisfied, a web service composition is synchronizable, i.e., it generates the same set of conversations under both the synchronous and the usual asynchronous communication semantics. LTL model checking can be conducted for such web service compositions using the synchronous communication semantics. Top-down conversation protocols can be analyzed in a similar way. A conversation protocol is realizable if there exists a web service composition which generates the same set of conversations as specified by the protocol. We propose several sufficient realizability conditions to restrict control flows of a conversation protocol so that realizability can be guaranteed. During the development of realizability analysis for conversation protocols, we have several interesting observations. For example, a conversation protocol is realizable if and only if it is realized by its projections to each peer. Realizability analysis on the FSA framework achieves better results than the Büchi framework – with two additional non-restrictive conditions we can guarantee freedom of unspecified message reception and freedom of deadlock in the FSA framework. The difference between the two models results from the inequivalence of nondetermin-

corresponding web service composition which realizes it; however, a web service composition does not always have a corresponding conversation protocol (using a finite state automaton) which specifies its conversation set.

istic and deterministic Büchi automata.

The synchronizability and realizability analyses have been extended to the Guarded Automata model where data semantics of web services are considered. We have developed a range of analysis techniques: a light-weight skeleton analysis which applies to the abstract control flows only, symbolic analyses for lossless join and synchronous compatibility, and iterative refined analysis for autonomy. Interestingly, in the Guarded Automata model, the bottom-up specification approach beats the top-down conversation protocol approach, for its much simpler symbolic and skeleton synchronizability analysis.

To specify real-world web service applications, and to support industry web service specification standards such as BPEL4WS and WSDL, a variation of the Guarded Automaton model, called XML-GA model, is developed to bring in XML data and XPath based data manipulation semantics. Formal models are established for XML related standards such as XML, (bounded) XML Schema, and a fragment of XPath query language. The XML-GA model is very expressive and allows transformation from most static web services that are specified using industry standards such as BPEL4WS.

In the Web Service Analysis Tool (WSAT), SPIN is employed as the back-end model checker. Translation algorithms are developed to translate from XML Schema to the type system of Promela, the input language of SPIN. Based on the type mapping, translation algorithms from XPath to Promela, and XML-GA to Promela are developed as well. A composition of interacting BPEL4WS web services can be translated into the XML-GA model, and then to a Promela specification, where LTL model checking is conducted. The ability to model and

verify XML data in WSAT allows to examine the correctness of web services at a great detail level, which helps to identify delicate bugs in a web service design.

8.1 Future Directions

Many interesting and immediate extensions of the Web Service Analysis Tool need to be explored. For example, we can greatly improve the verification speed via the implementation of symbolic verification modules into the tool (e.g. the BDD based symbolic model checking [18], and the Presburger arithmetic based infinite state symbolic verification [83, 17]). Many other model checking techniques such as predicate abstraction [47, 73, 29], counting abstraction [30], partial order reduction [46], and shape analysis [27] can also be implemented in WSAT.

In the long run, we plan to continue extending our automata-theoretic approach to the formal specification and verification of web services. In the following, we present some of the many possible future directions.

While the automata-theoretic models proposed in this dissertation have nicely captured the control flows and data semantics of *static* web services. Dynamic behaviors, e.g., dynamic instantiation of business processes and dynamic establishment of communication channels, are not addressed in the current model. We plan to incorporate these dynamic behaviors in the future. A number of questions that arise in the new model will be explored. For example, how do we specify a top-down conversation protocol for a web service composition where new peers can dynamically join? Will dynamic behaviors affect the realizability and synchronizability analyses? How do we verify systems with dynamic pro-

cess instantiations? Based on our previous experience with verifying Workflow systems [38], symbolic model checking plays a central role in reasoning about the infinite state space caused by dynamic process instantiations. This approach departs from the explicit state model checking techniques we currently use, and require further in-depth research work.

Currently, the XML-GA model in WSAT supports *bounded* XML data, where each XML context node can only have a bounded number of children nodes. In the future, we plan to investigate the use of tree-automata to encode XML documents with unbounded children nodes. It is interesting to explore how to extend the current tree-automata approaches [59, 66], to bring in the data semantics for XML leaf nodes. The automata based representation for arithmetic constraints [9] is a good starting point.

Since most web services in the real world are supported by back-end relational databases, one interesting problem is how to model and verify these web services as relational transducers. To verify such systems may require first order (or higher order) theorem prover. Although the general problem of deciding first order logic formula is undecidable, many decidable fragments exist. We are interested in studying these fragments and developing efficient decision procedures for them. Fast heuristic and approximation algorithms for the general problem also remain as one of our future research interests.

Bibliography

[1] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable specifications of reactive systems. In *Proc. of 16th Int. Colloq. on Automata, Languages and Programming*, volume 372 of *LNCS*, pages 1–17. Springer Verlag, 1989.

[2] P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. *Information and Computation*, 127(2):91–101, 1996.

[3] L. D. Alfaro and T. A. Henzinger. Interface automata. In *Proceedings 9th Annual Symp. on Foundations of Software Engineering (FSE)*, pages 109–120, 2001.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services Concepts, Architectures and Applications Series: Data-Centric Systems and Applications*. Addison Wesley Professional, 2002.

[5] Philippe Althern. The scala home page. `http://lamp.epfl.ch/scala/`.

[6] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. of the 22nd International Conference on Software Engineering (ICSE)*, 2000.

[7] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *Proc. 28th Int. Colloq. on Automata, Languages, and Programming*, 2001.

[8] R. Alur, K. McMillan, and D. Peled. Model-checking of correctness conditions for concurrent objects. *Information and Computation*, 160:167–188, 2000.

[9] C. Bartzis and T. Bultan. Automata-based representations for arithmetic constraints in automated verification. In *Proceedings of the Seventh International Conference on Implementation and Application of Automata (CIAA)*, 2002.

[10] R.V. Book and S.A. Greibach. Quasi-realtime languages. *Mathematical Systems Theory*, 4(2):97–111, 1970.

[11] Adam Bosworth. Loosely speaking. *XML and Web Services Magazine*, 3(4), April 2002.

[12] Business Process Execution Language for Web Services (BPEL4WS), version 1.1. *available at* `http://www.ibm.com/developerworks/library/ws-bpel`.

[13] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[14] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL a model for W3C XML Schema. In *Proc. of 10th World Wide Web Conference (WWW)*, pages 191–200, 2001.

[15] J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[16] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: A new approach to design and analysis of e-service composition. In *Proceedings of the Twelfth International World Wide Web Conference (WWW)*, pages 403–410, Budapest, Hungary, May 2003.

[17] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, pages 382–386, 2001.

[18] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[19] J. C.Corbett, M. B.Dwyer, J. Hatcliff, S. Laubach, C. S. Pasarenau, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proc. 22nd Int. Conf. on Software Engineering (ICSE)*, pages 439–448, 2000.

[20] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, and A. San giovanni Vincentelli. A formal specification model for hardware/software codesign. In *Proc. of the Intl. Workshop on Hardware-Software Codesign*, October 1993.

[21] A. Simon Christensen, A. Møler, and M. I. Schwartzbach. Extending java

for high-level web service construction. *ACM Trans. Program. Lang. Syst.*, 25(6):814–875, 2003.

[22] V. Christophides, R. Hull, G. Karvounarakis, A. Kumar, G. Tong, and M. Xiong. Beyond discrete e-services: Composing session-oriented services in telecommunications. In *Proceedings of Workshop on Technologies for E-Services (TES)*, Rome, Italy, September 2001.

[23] E.M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[24] OWL Services Coalition. OWL-S: Semantic markup for web services, November 2003.

[25] Collaxa.com. The Collaxa BPEL Server. `http://www.collaxa.com`.

[26] World Wide Web Consortium. Extensible markup language (XML). *available at* `http://www.w3c.org/XML`.

[27] J. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, January 2000.

[28] DAML-S (and OWL-S) 0.9 Draft Release. `http://www.daml.org/services/daml-s/0.9/`, May 2003.

[29] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proceedings of the 11th International Conference on Computer Aided Verification*, 1999.

[30] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer-Verlag, 2000.

[31] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier, 1990.

[32] Christopher Ferris and Joel Farrell. What are web services? *Comm. of the ACM*, 46(6):31–31, June 2003.

[33] A. Finkel and P. McKenzie. Verifying identical communicating processes is undecidable. *Theoretical Computer Science*, 174(1–2):217–230, 1997.

[34] P. C. Fischer, A. R. Meyer, and A. L. Rosenberg. Counter machines and counter languages. *Mathematical Systems Theory*, 2:265–283, 1968.

[35] D. Florescu, A. Grünhagen, and D. Kossmann. XL: An XML programming language for web service specification and composition. In *Proceedings International World Wide Web Conference (WWW)*, 2002.

[36] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering Conference (ASE)*, 2003.

[37] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and analysis of reactive electronic services. to appear in *Theoretical Computer Science*.

[38] X. Fu, T. Bultan, and J. Su. Formal verification of e-services and workflows. In *Proceedings of Workshop on "Web Services, e-Business, and the Semantic Web (WES): Foundations, Models, Architecture, Engineering and Applications"*, volume 2512 of *LNCS*, Toronto, Ontario, Canada, May 2002.

[39] X. Fu, T. Bultan, and J. Su. Conversation protocols: A formalism for specification and verification of reactive electronic services. In *Proc. 8th Int. Conf. on Implementation and Application of Automata (CIAA)*, volume 2759 of *LNCS*, pages 188–200, 2003.

[40] X. Fu, T. Bultan, and J. Su. Analysis of interacting web services. In *Proceedings of the 13th International World Wide Web Conference (WWW)*, pages 621 – 630, New York, May 2004.

[41] X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. To appear in the *2004 Int. Symp. on Software Testing and Analysis (ISSTA)* , 2004.

[42] X. Fu, T. Bultan, and J. Su. Realizability of conversation protocols with message contents. To appear in the *Proc. of 2004 IEEE Int. Conf. on Web Services (ICWS)*, 2004.

[43] X. Fu, T. Bultan, and J. Su. WSAT: A tool for formal analysis of web service compositions. To appear in the *Proc. of 16th Int. Conf. on Computer Aided Verification (CAV)*, 2004.

[44] X. Fu, O. Ibarra, J. Su, and T. Bultan. Message-based behavior models for web services: Expressive power. Manuscript, December 2004.

[45] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV 1995 Conference*, 1995.

[46] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of the 2nd Workshop on Computer Aided Verification*, LNCS 697, pages 438 – 449, 1993.

[47] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, June 1997.

[48] S. A. Greibach. Remarks on the complexity of nondeterministic counter languages. *Theoretical Computer Science*, 1(4):269–288, April 1976.

[49] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[50] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, 2003.

[51] IBM. Conversation Support Project. `http://www.research.ibm.com/convsupport/`.

[52] Java Message Service. `http://java.sun.com/products/jms/`.

[53] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of IFIP 74*, pages 471 – 475. North-Holland, 1974.

[54] Stans Kleijnen and Srikanth Raju. An open web services architecture. *ACM Queue*, 1(1):39–46, March 2003.

[55] M. Koshkina and F. van Breugel. Verification of business processes for web services. Technical report, York University, 2003.

[56] H. Liu and R. E. Miller. Generalized fair reachability analysis for cyclic protocols. In *IEEE/ACM Transactions on Networking*, pages 192–204, 1996.

[57] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings 6th ACM Symp. Principles of Distributed Computing*, pages 137–151, 1987.

[58] M. Machtey and P. Young. *An Introduction to the General Theory of Algorithms*. North Holland, New York, 1979.

[59] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. of 21th Symposium on Principles of Database Systems (PODS)*, pages 65–76, 2002.

[60] Gerry Miller. .Net vs. J2EE. *Comm. of the ACM*, 46(6):64–67, June 2003.

[61] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[62] A. Møller. Document Structure Description 2.0, December 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from `http://www.brics.dk/DSD/`.

[63] Message Sequence Chart (MSC). ITU-T, Geneva Recommendation Z.120, 1994.

[64] Microsoft Message Queuing Service. `http://www.microsoft.com/msmq/`.

[65] S. Narayanan and S. A. Mcllraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[66] F. Neven. Automata theory for XML researchers. *Sigmod Record*, 31(3), 2002.

[67] Eric Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Springer, 2004.

[68] Object Management Group (OMG). Common object request broker architecture (corba), July 1995.

[69] W. Peng. Single-link and time communicating finite state machines. In *Proc. of 2nd Int. Conf. on Network Protocols (ICNP)*, pages 126–133, 1994.

[70] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of 16th ACM Symp. Principles of Programming Languages*, pages 179–190, 1989.

[71] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *Proc. of 16th Int. Colloq. on Automata, Languages, and Programs*, volume 372 of *LNCS*, pages 652–671, 1989.

[72] S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *Proc. of Static Analysis Symposium (SAS)*, July 2001.

[73] Hassen Saidi. Model checking guided abstraction and analysis. In Jens Palsberg, editor, *Seventh International Static Analysis Symposium (SAS)*,

volume 1824 of *Lecture Notes in Computer Science*, pages 377–339, Santa Barbara, CA, July 2000. Springer-Verlag.

[74] Simple Object Access Protocol (SOAP) 1.1. W3C Note 08, May 2000. (`http://www.w3.org/TR/SOAP/`).

[75] Universal Description, Discovery and Integration (UDDI) protoocl. `http://www.uddi.org`.

[76] IBM WebSphere software platform. *available at* `http://www-306.ibm.com/software/info1/websphere/index.jsp`.

[77] Joseph Williams. The Web Services Debate J2EE vs. .Net. *Comm. of the ACM*, 46(6):59–63, June 2003.

[78] Web Service Analysis Tool (WSAT). `http://www.cs.ucsb.edu/~su/WSAT`.

[79] Web Service Choreography Interface (WSCI) 1.0. `http://www.w3.org/TR/2002/NOTE-wsci-20020808/`, August 2002.

[80] Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001.

[81] XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, 1999.

[82] XML Schema. *available at* http://www.w3c.org/XML/Schema.

[83] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume

2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.