AUTOMATIC GENERATION OF PROTOCOL CONVERTERS FROM SCENARIO-BASED SPECIFICATIONS

TRAN TUAN ANH

(B.Comp.(Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgements

I would like to thank my supervisors Professor P.S. Thiagarajan and Dr Abhik Roychoudhury for their guidance, help, and advice. I have been working with Prof. Thiagu - as we usually call him - since my honour year thesis; and certainly I have learned a lot from his experience and wits. Needless to say, without my supervisors this thesis will not be possible.

This thesis is a landmark in my life. It marks the end of my six years in NUS, and it marks the end of the first and the longest study period of my life - since primary school till now. Finally, I go out to the "real" world to contribute my little labor ability to the society. Hence, this thesis marks a new journey. I take this chance to thank National University of Singapore, in particular, and The Ministry of Education, in general, for the generous scholarship supports I have received during my study years in NUS.

I thank my friends here in Singapore who in one way or the other help me to "endure" this tough period. I name a few (never meant to be complete): William Hung - I seriously think you should drop the PhD and become a writer, Dung Tri - one of the nicest guys I have ever met, Viet Dung - a (almost) Kung Fu master and the most weird lawyer, Thanh Liem - no one can be more “child-ful” than him, Minh Hang - our model who gives us so much laughter, Tran Toan - our beloved poet who without him cooking is less fun since you have to care about cleaning, Thang - who shows me how simple life can be.

My life in Singapore has many good memories with the old generation of Vietnamese students in NUS - Chi, Mai Anh, Tuan Den, Dung Den, Dung Trang, Son, Hieu, Hoai Thu, etc. I think we have shared a lot of difficulties and also many joyful moments - for most of us those years are the first time we were away from family. Thanks all of you for the wonderful old days.

I should mention Trung, Tuan, Mai Lam - my buddies since high school. I am blessed to have such friends. I always can pass by their houses, get them on a bike, and end up siting in a decade-old coffee shop in the old quarter. Best friends!

Things happen, people change; the only constant in life. However, one thing never changes is our family. They are always there for you; successes or failures they share with you. I had a wonderful childhood yet sharing the most difficult time of our family with my mum. She is the most sober woman I know. In the family, I have a “buddy” whom is uncle Chuong. You cannot ask more from life. I am blessed!

I cannot thank enough my girl-friend Hong Ha for her companionship. It has been a long time since we first met each other - well in Singapore, and it has been a long happy time. You are always here when I need you! And as always through you I get to know our Hanoi much more!!!

Tran Tuan Anh

June 2005

Contents

Summary

In the last 20 years, embedded computing systems have become most prevalent carriers of advanced hardware and software technologies. The use of embedded systems in many common place applications and household products requires more stringent requirements not normally expected from traditional computer systems. Additional, product cycles continue to shrink as evidenced by the adoption rate of consumer electronic items. This has fueled the needs for better methodologies and tools to design, analyze, implement, and deploy such systems.

The implementation of embedded systems has evolved from using micro-controllers and discrete components to fully integrated systems-on-chip (SoC). Leading edge SoCs being designed today could reach 20 million gates and 0.5 to 1 GHz operating frequency. Consequently, it is impossible for a single company to design and manufacture an entire electronic system in time and within reasonable cost. Hence, design re-use based on pre-designed intellectual property (IP) cores has become an absolute necessity for embedded system companies to remain competitive. Since IP cores are pre-designed and pre-verified, the designer will be able, in principle, concentrate on the whole system at a high level without having to worry about the correctness and performance of the individual components. However, the vision of quickly assembling an SoC using IP cores has not yet become reality for various reasons. One of the difficulties in IP reuse is the incompatibilities between the protocols used by various parts. Hence, reusing IP cores often requires designing converters (glue-logic) to enable their communication.

In this work, we study the problem of automatically generating a protocol converter which enables various embedded system components - using different (possibly incompatible) protocols - to talk to each other. The novelty of our approach is that it takes as input a scenario-based description of inter-component interactions described as a collection of Message Sequence Charts. From this specification, we systematically synthesize, when possible, the protocol converter that lets the components to use their native protocols while overall pattern of interaction is correctly realized. Both of the input component protocols and the synthesized protocol converter are described in SystemC hence we are able to compile the converter along with the component protocols. The result system can be simulated using the SystemC kernel. Our work is not restricted to uni-directional communication, and the converter can be used to broker communication among many components. We demonstrate the feasibility of our approach by modeling some important features of existing Systems-on-Chip bus protocols.

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the last 20 years, embedded computing systems have become prevalent. More than 90 percent of existing computer systems are embedded systems consisting of both hardware and software components. As we are marching forward in the 21 century, embedded systems will surely play a dominant role; they will be present into every aspect of our daily life.

The use of embedded systems in many common place applications and household products requires more stringent requirements not normally expected from traditional computer systems. These applications demand high performance, power-consciousness, high reliability and predictability. Additionally, product cycles continue to shrink as evidenced by the adoption rate of consumer electronic items. This has fueled the needs for better methodologies and tools to design, analyze, implement and deploy such systems.

IP Reuse The implementation of embedded systems has evolved from using micro-controllers and discrete components to fully integrated systems-on-chip (SoC). The concept of systems-on-chip is to integrate all components on a board into a single chip. Designing a SoC, however, is extremely complex. Leading-edge SoCs being designed today could reach 20 million gates and 0.5 to 1 GHz operating frequency. Consequently, it is impossible for a single company to design and manufacture an entire electronic system in time and within reasonable cost.

As a result, design re-use based on pre-designed intellectual property (IP) cores has become an absolute necessity for embedded system companies to remain competitive. A new industry has evolved to devote solely to the development of IP cores as reusable building blocks for SoCs. Since IP cores are pre-designed and pre-verified, the designer is in a better position to concentrate on the complete system without having to worry about the correctness and performance of the individual components.

In practice, however, the vision of quickly assembling an SoC using IP cores has not yet become reality for various reasons. Some of the reasons have been identified in (Bergamaschi and Lee, 2000); they vary from the need to understand electrical characteristics of components to complexities of system verification. We highlight some main difficulties here:

- The designers need to fully understand the functionality, *interfaces* and electrical characteristics of complex IP cores.
- Even if the cores are pre-verified, it does not mean the whole system will work when they are put together. *Various interface and timing issues* can cause systems to fail even though the individual cores are correct.
- The lack of the established *standard deliverables* and the lack of efficient *interface synthesis tools* make it difficult for IPs from different providers to be integrated into the same SoC.

We can see that the incompatibility between multiple interface protocols and non-standard specifications are among the main difficulties of the IP-based SoC design.

System Level Design The complexity of the current systems and the need to increase the productivity require us to raise the level of abstraction in which SoC designs are performed. Traditionally, computer-aided-design tools have focused on low-level design issues, such as synthesis, timing, layout and simulation. Recently, modeling approaches using system-level design languages have been developed (Arnout, 2000; Flake and Davidmann, 2000; SystemC; SystemVerilog, 2005;

Website, 2005). One advantage of system-level design-languages is that it encourages the incorporation of the IP cores from various sources. All of IP cores developed in different companies need to use a common system-level design language so that the entire system can be modeled in a single framework. Without a standard system-level design language, IP vendors are forced to choose the language(s) they want to support to describe IP interfaces - IP designs are, in general, not available. The designer is then faced with integrating IP cores described in different and incompatible modeling languages. This is a barrier to realize the IP-based design approach. Once we have a common system level design language, one way to help to build an SoC by integrating and being configured IP cores easily is to provide high-level tools to automate the integration of IP cores with different interfaces and communication protocols.

Hardware-Software Codesign Another source for the need of interface synthesis tools is due to Hardware-Software Codesign. In the recent past, significant effort in embedded systems research communities has been put in this topic (Chinook; Polis). The problem here is to coordinate the design of the parts of the system to be implemented as software and the parts to be implemented as hardware, avoiding the HW/SW integration problem. The system is first specified in terms of functionalities, this can be done in formal models or systems level design languages. After that the system is partitioned into components to map to hardware/software blocks. The HW-SW partitioning decisions are based heavily on design experience and very difficult to automate. The designer needs an environment in which evaluation of such decisions in terms of various criteria can be done easily and quickly. In the HW-SW codesign paradigm, a component which is first mapped to hardware for performance gain can latter be implemented as a software component for flexibility. This introduces the need for an automatic way to synthesize the interfaces between components. We have to deal with three kinds of interfaces: interfaces between software and software components, between hardware and hardware components, and between software and hardware components. It is often to be seen that the interfaces between software and hardware components is harder to synthesize than other twos.

Intra- and Inter- view in System Level Design In order to cope with the complexity of the current systems, an important component of a new system design paradigm is the orthogonalization of concerns, i.e., the separation of the various aspects of design (Gajski, Zhu, and Domer, 1997; Grotker, Liao, Martin, and Swan, 2002; Metropolis Project; Sangiovanni-Vincentelli, Sgroi, and Lavagno, 2000). The two common ones are:

- the separation between function (what the system is supposed to do) and architecture (how the system is implemented);
- the separation between computation and communication.

The second point above plays an important role in promoting the IP-based design. If in a design component behaviors (computation) and communications are tightly coupled, it is very difficult to re-use components since their behaviors are dependent on the communication mechanisms of other components of the original design. If we can achieve the separation between computation and communication at the functional level, the designer will be able to easily try out various implementation solutions for each component which in turn will promotes the IP-based design paradigm. Hence it is fruitful to have two dual views of the overall system:

- An intra-component view where one provides *per component* its computational and control flow with its communication activities abstracted as atomic actions; each such action will stand for a possibly complex interaction with other components.

- An inter-component view which suppresses the computational aspects of the individual components and instead provides a global specification of the interaction patterns that need to be realized.

The relationship between intra-component and inter-component (scenario-based) system descriptions is complex and subtle (Harel and Marelly, 2003). We feel that, from a pragmatic standpoint, it is best to have both forms of description. Of course, this raises the issue whether the two dual views of the same system are consistent. We feel this question can be settled given the rich body of results concerning MSCs that are available (see for instance, the survey (Harel and Thiagarajan, 2003) and the references therein).

In conclusion, the wide spread of embedded systems with ever increase of their complexity make design re-use based on IP cores an absolute necessity. To facilitate the effective reuse of IP cores, we need standard languages for the documentation and interface specification, along with methods for checking the compatibility of components in a design. The modeling languages to be used must be flexible enough to be able to describe both hardware and software components. *Moreover, there must be automatic tools for synthesizing the component adaptors to bridge between incompatible interfaces.* In the task of bridging the interface incompatibilities, we deal mostly with the interaction patterns between blocks; the inter-component view of the system can be extremely useful here as we demonstrate later on.

1.2 Protocol Converter Synthesis Problem

Protocol converter synthesis seeks to automate the process of interconnecting components. A major problem here is the realization of the interconnect fabric; the system will consist of multiple components, typically supplied by different vendors that will have to communicate with each other in specified patterns. However, the protocols assumed by the different components are often incompatible. For instance, Figure 1.1 shows an example in which a sender would like to send a data token to a receiver. The sender IP core has been designed on the basis of the *Ack-Nack* protocol for transferring data in which it keeps trying to transmit data message *msg* until it gets back a *ack* signal, whereas the receiver IP core has been designed on the basis of the *Pull-End* protocol where it sends out a *pull* signal followed by the reception *data* message and a *end* signal. If we let these two components talk to each other directly, they ,obviously, cannot communicate. Hence, we need a glue logic which enables such pairs of incompatible components to communicate. The role of this glue logic is to let the components to use their native protocols and be oblivious to the incompatibility of the protocol being used by other parties.

For instance, in the example in Figure 1.1, we can insert between the sender and the receiver a glue logic which will do the following:

- Receive the *msg* signal from the sender,
- Receive the *pull* signal from the receiver,
- Translate the *msg* message to the *data* message and send *data* to the receiver,
- Send *ack* signal to sender and *end* signal to receiver.

This simple glue logic will enable the sender and the receiver to communicate using their own protocols. So the system becomes like in Figure 1.2. The main job of the glue logic here is to *convert* one protocol to the other, hence we call it a protocol converter. There are many levels of interconnection that must be considered including electrical, power, logic, register-transfer, device drivers, and higher software levels (Borriello, Lavagno, and Ortega, 1998). We are particularly interested in communication protocols in system level designs of systems-on-chip.

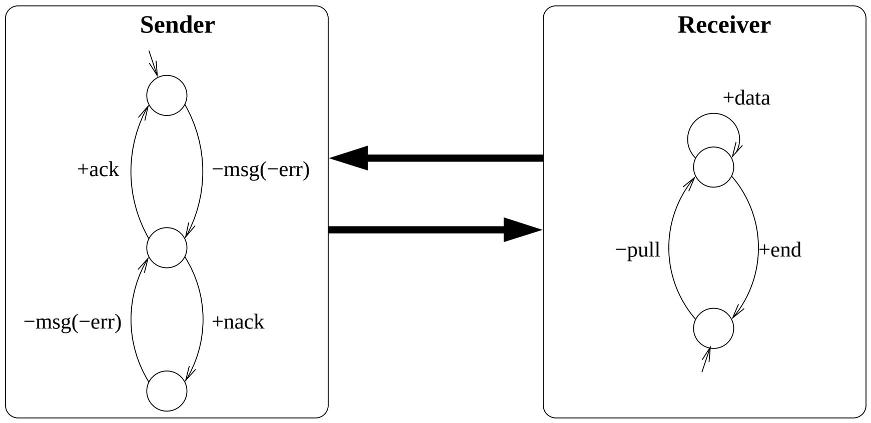


Figure 1.1: A pair of incompatible protocols: the sender using the Ack-Nack protocol and the receiver using the Pull-End protocol. Outgoing messages are denoted with the negative sign; for instance, $-msg$ and $-pull$. Incoming messages are denoted with the positive sign; for instance, $+nack$ and $+end$. The notation is from the viewpoint of the protocols involved.

1.3 Our Approach

In this thesis, we investigate the use of *scenarios* for the specification and realization of protocol converters. We show how the popular visual formalism of Message Sequence Charts (MSCs) and High-level Message Sequence Charts (HMSCs) (Z.120, 1996) can be used to specify communication patterns, in which each of the participating component has its own view of the protocols being deployed. From this specification, one can systematically synthesize, where possible, the *protocol converter* that lets the components to use their native protocols while the overall pattern of interactions is correctly realized. We say “where possible” because there will be native protocols that are inherently incompatible, and in this case our synthesis method will abort after detecting and declaring this incompatibility.

In the inter-component description of a system, we view an MSC as an atomic

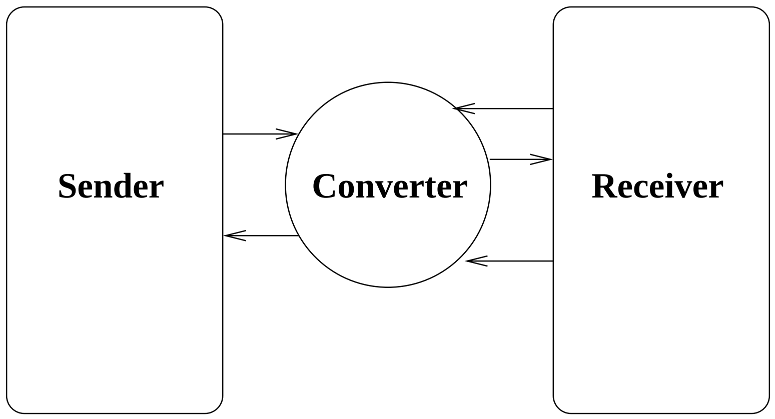


Figure 1.2: The role of the protocol converter

unit of interaction involving two or more components. Even when there are only two components- which will often be the case- the flow of information will not be unidirectional. Data might flow only one way but control signals will typically flow both ways. An interaction pattern will consist of a (concatenated) sequence of MSCs and the set of all such patterns of interest will be captured by an HMSC. An HMSC is a standard way of specifying a collection of MSCs. It is basically a finite state automaton which has an MSC associated with each state. Each sequence of states allowed by the automaton will induce a sequence of MSCs whose concatenation, will yield an MSC. The collection of all such MSCs is the set of interaction patterns specified by the HMSC.

In our setting, each node of the HMSC represents a ***mode of interaction***, that is, a snippet of a protocol. Consequently, each node of the HMSC will have , in general, not just one MSC associated with it. Instead, it has a *set* of associated MSCs: one corresponding to the "view" (or the native protocol) of each component taking part in this snippet. Figure 1.3 shows our HMSC description of the interaction between two protocols described in Figure 1.1. The two components go through various *modes of interaction* or phases. In each phase, each component has its own view of the communication pattern. A node will typically contain two MSCs Ch_m and Ch_s; Ch_m describes the *master*'s view and Ch_s captures the *slave* component's view of *transfer*. For example, the *sender* using the Ack-Nack protocol does not have set-up phase, hence the MSC describing its view in the *set-up* node is empty. On the other hand, the MSC describing the receiver's view in the *error-recovery node* is empty, since the Pull-End protocol does not have this function.

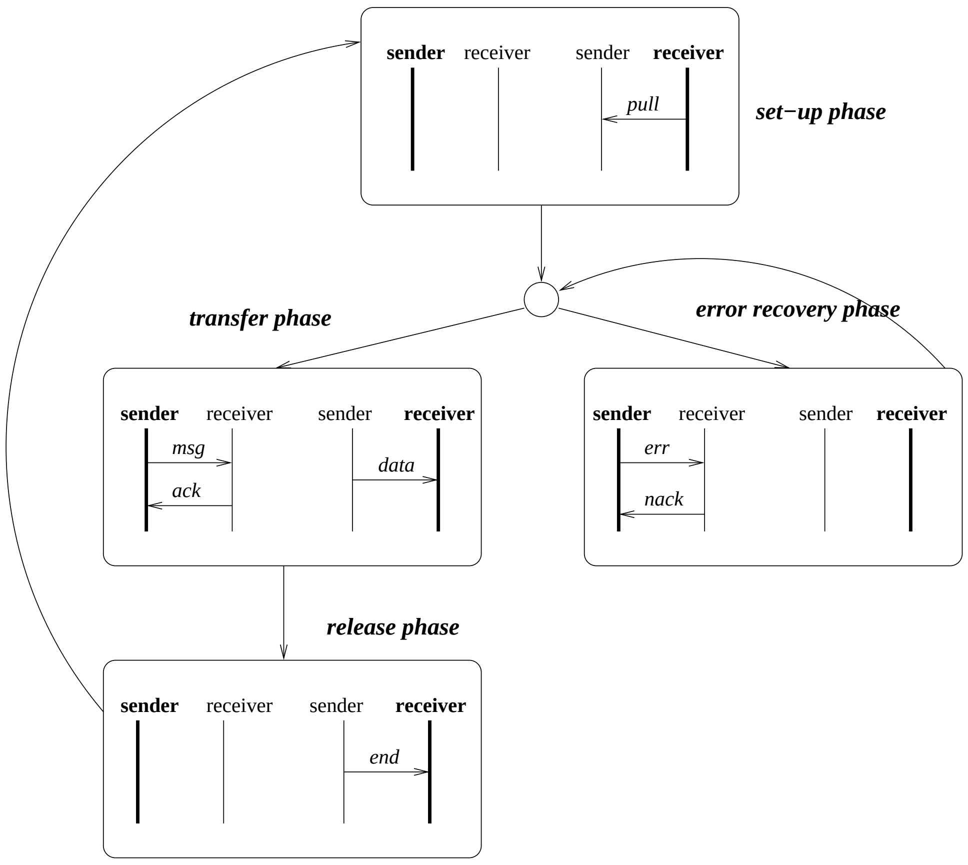


Figure 1.3: An HMSC description of the interaction between incompatible protocols

Technical Contributions Our goal is to design a converter that will realize all possible interaction patterns (*i.e.* *runs*) of a given HMSC while permitting all the components involved to execute their views of the modes of interaction they participate in. Our converter will sit in the midst of the components. All the signals/data sent by the components will flow to the converter and all the signals/data received by the components will be generated by the converter.

Our converter will deploy a number of techniques in the attempt to smooth out incompatibilities between different views of the components. First, it will generate or consume control signals. We allow the converter to speculatively generate control signals in advance which might resolve potential deadlocks. However, as might be expected, the converter will not generate data values speculatively. Secondly, for messages which carry data values a message relationship specification is given to guide the converter to translate and map messages across components. The converter is able to deal with data format incompatibilities such as 16 bit sends vs. 8 bit receives etc. In this process, the converter will store data in order to resolve incompatibilities. Lastly, in certain settings, the converter should not generate speculatively control signals; for example, when these signals are a part of a protocol ensuring mutual exclusion. To cope with this situation, we allow the designer to specify, in addition of HMSC specification, additional behavioral requirements to have additional control on the behavior of the protocol converter.

We have implemented our technique using SystemC. The (textual) input to our converter generator is an HMSC with multiple MSCs associated with each node of the HMSC; one for each component taking part in the mode represented by the node. This input together with the behavioral requirements and the message relationship specifications are used to generate a SystemC implementation of the converter, when one exists. The validation of the converter is carried out by supplying a path through the HMSC using which the SystemC simulator will execute the converter and display the resulting run in the form of an MSC. Due to translation into SystemC we have been able to introduce clock sensitivities and timers in our input specification language.

1.4 Thesis Organization

The rest of this thesis is organized as follows. In the next chapter, we briefly recall the basic features of MSCs and HMSCs together with their precise formal semantics. In chapter 3, we survey related work on the protocol converter generation problem. In chapter 4, we formulate the protocol converter generation problem and describe our solution in details. The next two chapters provide information about our implementation and the applications of our technique. Chapter 7 concludes the thesis with a summary of the contribution of this thesis and a discussion on directions for future research.

Chapter 2

Preliminaries

We present in this chapter the basic notions and formalism concerning MSCs and HMSCs. For a more detailed survey, the reader is referred to (Harel and Thiagarajan, 2003).

2.1 Message Sequence Charts

An MSC describes a snippet of behavior involving multiple components interacting with each other. In its simplest form (which is the one we use here), the components communicate with each other through FIFOs. The visual representation of an example MSC is shown in Figure 2.1. The vertical lines, often referred to as lifelines or instances, capture the behavior of the components. A horizontal edge captures a communication; the origin of the edge is the sender, the target of the edge is the receiver, and the label associated with the edge is the message being communicated. We adopt the usual MSC convention that horizontal edges can be drawn either horizontally or sloping downwards, but not upwards. The darkened rectangular boxes are the events associated with the MSC. In the following we will often refer to an MSC as a chart.

The example shown in Figure 2.1 describes a scenario in which a user U sends a request to an interface I to gain access to a resource R. The interface in turn sends a request to the resource and receives *grant* as a response, after which it sends *yes* to U. The internal event labeled *count* may involve the interface component incrementing the variable to track the number of times the user has gained access to R.

Next, we give the formal definition of MSCs. We fix a finite set of processes (or components) $\mathcal{P}$ and let p,q,r range over $\mathcal{P}$. We shall use Σ_p to denote the set of actions executed by the process p. We define $\Sigma_p = \{\langle p!q, m\rangle \mid p \neq q \text{ and } m \in M\} \cup \{\langle p?q, m\rangle \mid p \neq q \text{ and } m \in M\} \cup \{\langle p, a\rangle \mid a \in \mathit{Act}\}$ where M is an alphabet

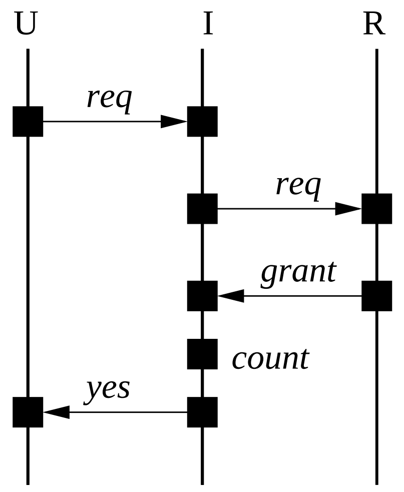


Figure 2.1: A Simple Message Sequence Chart.

of messages and Act is an alphabet of internal actions. The communication action $\langle p!q, m\rangle$ is to be read as p sends the message m to q, and the communication action $\langle p?q, m\rangle$ is to be read as p receives the message m from q. On the other hand, the internal action $\langle p, a\rangle$ stands for p executes action a. We set $\Sigma = \bigcup_{p\in\mathcal{P}} \Sigma_p$. We also denote the set of *channels* by $Ch = \{(p, q) \mid p \neq q\}$ and let c,d range over Ch.

Turning now to the definition of MSCs, we first define a Σ-labeled poset to be a structure $M = (E, \preceq, \lambda)$ where $(E, \preceq)$ is a poset and $\lambda : E \to \Sigma$ is a labeling function. For $e \in E$ we define $\downarrow e = \{e' \mid e' \preceq e\}$. For $p \in \mathcal{P}$, we set $E_p = \{e \mid \lambda(e) \in \Sigma_p\}$, these are events that p takes part in. Furthermore, $E_{p!q} = \{e \mid e \in E_p$ and $\lambda(e) = \langle p!q, m\rangle\}$ for some m in M, $E_{p?q} = \{e \mid e \in E_p$ and $\lambda(e) = \langle p?q, m\rangle\}$ for some m in M. For each $c \in Ch$, we define the *communication relation* $R_c = \{(e, e') \mid \lambda(e) = (p!q, m)$ and $\lambda(e') = (q?p, m)$ and $|\downarrow e \cap E_{p!q}| = |\downarrow e' \cap E_{q?p}|\}$. Finally, for each $p \in \mathcal{P}$, we define the *p-causality relation* $R_p = (E_p \times E_p) \cap \preceq$.

Definition 1 (Message Sequence Charts). An MSC over $(\mathcal{P}, M, Act)$ is a *finite* Σ-labeled poset $M = (E, \preceq, \lambda)$ which satisfies the following conditions:

1. Each R_p is a linear order,

2. For every p, q with $p \neq q$, $|E_{p!q} = E_{q?p}|$,

3. Suppose $e \in E_{p?q}$. Then $| \downarrow e \cap E_{p?q}| = | \downarrow e \cap E_{q!p}|$,

4. Suppose $\lambda(e) = \langle p!q, m \rangle$ and $\lambda(e') = \langle q?p, m' \rangle$ and $| \downarrow e \cap E_{p!q}| = | \downarrow e' \cap E_{q?p}|$. Then $m = m'$.

5. $\preceq = (R_{\mathcal{P}} \cup R_{Ch})^*$ where $R_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} R_p$ and $R_{Ch} = \bigcup_{c \in Ch} R_c$.

The first condition says that all the events that a process takes part in are linearly ordered; each process is sequential. The second condition says that there are no dangling communication edges in an MSC; the number of sent message must be equal to the number of received messages. The third condition says that messages must be sent before they can be received; the order of communication actions must be correct. Furthermore, the fourth condition requires that the message names must be correctly ordered. Lastly, the partial order of an MSC is its *visual order*; This partial order is the transitive closure of (a) the total order of the events in each process (time flows from top to bottom in each process) and (b) the ordering imposed by the send-receive of each message (the send event of a message must happen before its receive event). Semantically, a chart denotes a set of events (message send, message receive and internal events corresponding to computation) and prescribes a partial order over these events. Any sequence of these events in which each event of the chart occurs exactly once and in which the order of appearances of the events respects this causal order will be called a linearization of the chart. Each linearization constitutes *an execution of the chart*.

2.2 High-level Message Sequence Charts

One standard mechanism (Z.120, 1996) for presenting a *collection* of MSCs is called high-level MSCs (HMSCs). An HMSC is basically a finite state automaton whose states are labeled by MSCs. Consequently, one can write out specifications involving choice, concatenation and iteration operations over a finite set of seed MSCs. In general, a HMSC specification can be hierarchical in which a state of the automaton can be labeled by an HMSC instead of an MSC. In this thesis, we shall ignore this feature and instead consider flattened HMSCs.

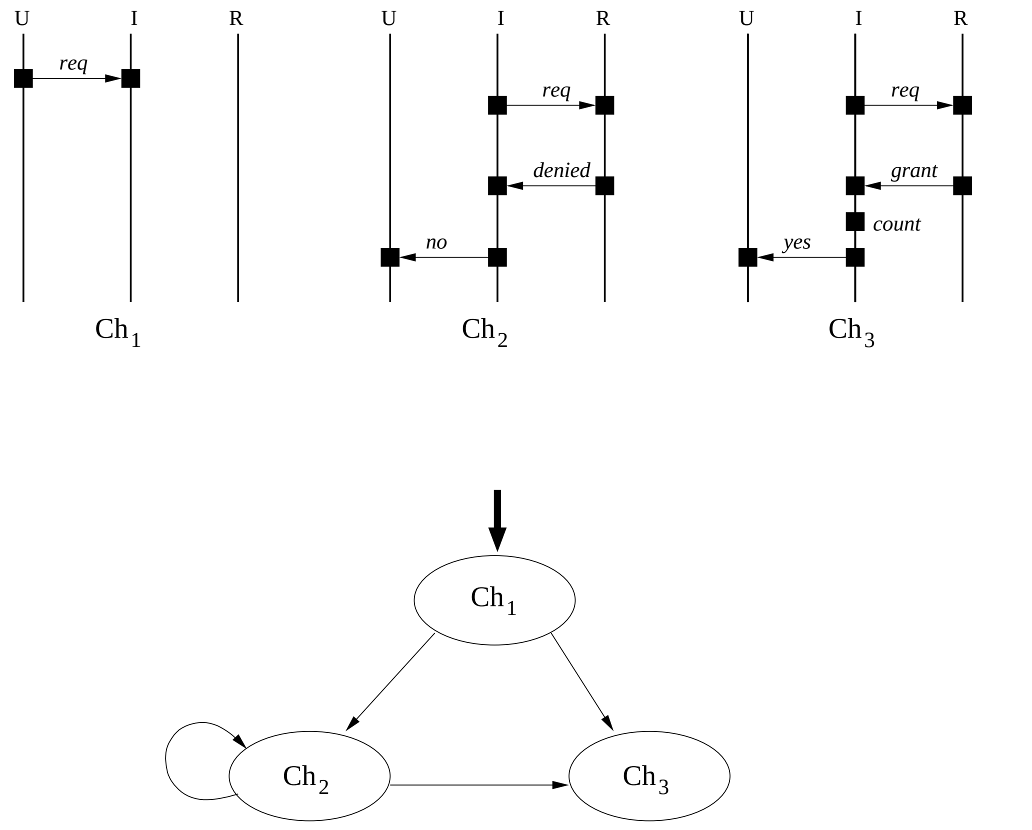


Figure 2.2: A Simple HMSC

An example of an HMSC is shown in Figure 2.2. Intuitively, this HMSC captures a collection of scenarios consisting of a user U sending a request to an interface I to access a resource R. The interface queries the resource, and if it gets the response *denied*, it sends a *no* to the user and tries again. It will keep trying until it gets the response *granted*, at that point it send *yes* to U, and the transaction ends.

The edges in an HMSC represent the natural operation of chart concatenation. The collection of charts represented by an HMSC consists of all those charts obtained by tracing a path in the HMSC from an initial control state to a terminal control state and concatenating the MSCs that are encountered along the path. In our setting, the terminal states will not be important. Hence we will assume that by default all the states are terminal states. There are two intuitive ways of concatenating charts. In synchronous concatenation, $Ch_1 \cdot Ch_2$ denotes a scenario in which *all* the events in Ch_1 must finish before *any* event in Ch_2 can occur. Thus synchronous concatenation requires all the concerned life-lines to synchronize at the end of any MSC. It rules out the parallelism that could arise had we let the second chart start its operation before the predecessor chart has completely finished.

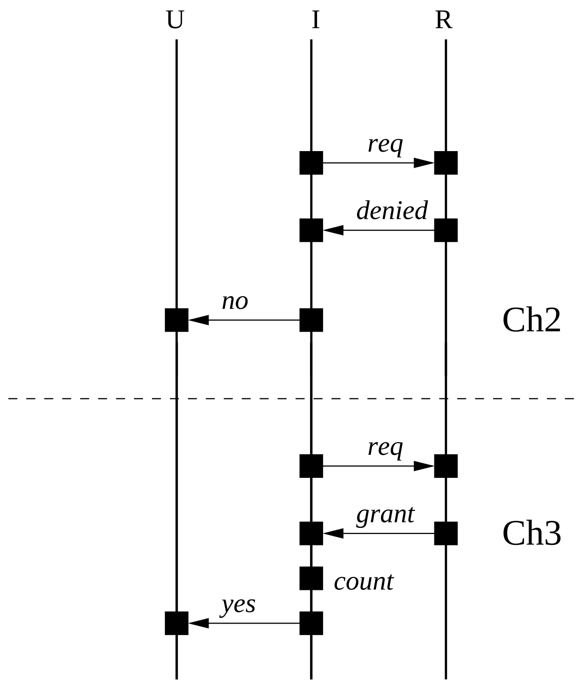


Figure 2.3: Concatenation of MSCs

The second way of concatenating charts - which is the one we will consider in this thesis - is the asynchronous concatenation. Here the concatenation is carried out at the level of life-lines. In Figure 2.3 we show the chart Ch_{23} obtained via the asynchronous concatenation $Ch_2 \circ Ch_3$ where Ch_2 and Ch_3 are as shown in Figure 2.2. Note that the receipt of `no` in Ch_2 can take place after the sending of `req` in Ch_3 under asynchronous concatenation.

More formally, let $Ch^1 = (E^1, \preceq^1, \lambda^1)$ and $Ch^2 = (E^2, \preceq^2, \lambda^2)$ be a pair of MSCs. Assume that E^1 and E^2 are disjoint sets. Then $Ch^1 \circ Ch^2$ is the MSC $(E, \preceq, \lambda)$, where:

- $E = E^1 \cup E^2$
- $\lambda(e) = \lambda^1(e)(\lambda^2(e))$ if e is in $E^1(E^2)$
- $\preceq$ is the minimal ordering relation over E that contains $\preceq^1$ and $\preceq^2$ and satisfies: If $e \in E^1_p$ and $e' \in E^2_p$ for some p, then $e \preceq e'$.

Recall that the meaning of an HMSC is given by a -potentially infinite- collection of MSCs; these are generated from the paths from an initial state (to a final state) in the graph - again, we assume all states are final. For each such path, we asynchronously concatenate the induced sequence of MSCs, and the resulting MSC is in the collection represented by the HMSC. Thus, for the HMSC in Figure 2.2, the chart $Ch_1 \circ Ch_2 \circ Ch_2 \circ Ch_3$ is in the collection while $Ch_1 \circ Ch_3 \circ Ch_2$ is not.

We conclude by introducing the notion of cycle bounded executions of an HMSC. This notion will be used to restrict the amount of overtaking allowed between processes in our SystemC implementation. Suppose

$$s_0 s_1 \ldots s_i s_{i+1} s_{i+2} \ldots s_m$$

is a path in an HMSC. In other words s_0 is an initial state and (s_j, s_{j+1}) is an edge in the HMSC for $0 \leq j < m$. Suppose now further $s_i = s_{n+i}$ with $0 \leq i < n+i \leq m$

so that $\pi = s_i s_{i+1} \ldots s_{n+i}$ constitutes a cycle in the HMSC. Let Ch_j be the chart associated with s_j for each j in $\{0, 1, \ldots n+i\}$ and $Ch = Ch_0 \circ Ch_1 \circ \ldots Ch_m$. We will say that the execution σ of Ch is π-bounded iff all the events of Ch_i appear in σ before any event of Ch_{n+i} appears in σ. Now let Ch be an MSC generated by an HMSC and σ an execution of Ch. We will say that σ is *cycle-bounded* if it is π-bounded for every cycle π contained in the path that generates Ch.

2.3 Notational Conventions

Our technique for automatically generating protocol converters takes in as input an extension of the HMSC description of the interaction among a network of components. In our extended HMSC description, each node contains a set of MSCs; one MSC for each component taking part in the corresponding mode of interaction associated with that state. Figure 1.3 shows an example of our extended HMSCs. For convenience, we will refer to this extended notation too as an HMSC.

Chapter 3

Literature Review

3.1 Related Work

The problem of generating glue logic for protocol conversion has been studied in the past. There are many levels of interconnection that need glue logic including electrical, power, logic, register-transfer, device drivers, and higher software levels (Borriello et al., 1998). In this chapter, we survey the previous work on the problem of automatically generating protocol converters. In particular, we are interested in communication protocols in system level designs of systems-on-chip.

Borriello (1988) generated glue logic between two circuit blocks, whose interface behavior is captured by timing diagrams, via transducer synthesis. The work of (Akella and McMillan, 1991) describes protocols as finite state machines and develops the protocol converter from the product machine. Narayan and Gajski (1995) develop a protocol converter from the HDL description of two component protocols; a nice feature of this work is the ease of simulation of the protocol converter along with the component interfaces. These works generate a protocol converter for enabling communication among two components; on the other hand, our work focuses on generating a converter to broker communication among multiple components.

To the best of our knowledge, our work is the first one to study protocol converters using scenario-based descriptions. Previous works that have influenced our research have been carried out in intra-component settings (de Alfaro and Henzinger, 2001; Passerone, de Alfaro, Henzinger, and Sangiovanni-Vincentelli, 2002; Passerone, Rowson, and Sangiovanni-Vincentelli, 1998). In these works, one describes the protocol behavior of each component using automata, or equivalently, regular expressions. In doing so, one abstracts away the internal computational aspects of the components and focuses mainly on the flow of signals and data across the communication interface of the component.

In (Passerone et al., 1998) the authors address the problem of synthesizing protocol converters between communicating components that use different signaling conventions. More specifically, given two protocol descriptions of two components, an algorithm is proposed to build an protocol converter (interface machine in their terms) so that data transfers are consistent with both protocols. The focus here is on a *single* transaction (i.e. roughly corresponding to a single MSC in our terms) with all information flowing one way between two components, and the component protocols are described using regular expressions. There are few simplifying assumptions as follows.

- The generated converter is only for *point-to-point* communication between two components.
- Both components agree on a composite data structure called *token*, and all the data transfers between two components are referred to this *token*.
- The converter can temporarily store the data, and it has enough memory to store the whole token which is all the data which need transferring.
- Components are fully synchronous by the same clock.

The objective of the synthesis algorithm is to produce a finite state machine that when placed between the two modules implementing the specified protocols it will make the communication possible. The interface recognizes symbols on the producer side and generates proper symbols on the receiver side. The algorithm given in paper can be divided into three steps. First, the regular expressions representing two protocols are translated into automata. After that these two automata are composed to get the product finite state machine. The alphabet of the product machine is the Cartesian product of the two alphabets. Signals from the producer become inputs, signals from the receiver become outputs. Finally, the product machine is reduced so that it only contains legal sequences of operations. Any non-determinism is also resolved according to the following rules: 1) never output a piece of data that has not been received 2) transfer the data 3) minimize the latency.

The authors mentioned some of interesting extensions: asynchronous protocols, many parties in the communication, and reactive components (not just a single transaction). The reactive component extension is later tackled in (Passerone et al., 2002). However, it is not clear how to extend the approach in this paper for situations with more than two components.

de Alfaro and Henzinger (2001) proposed to use automata to capture the *temporal* aspects of *software* component interfaces. The formalism is called *interface automata*. Since the components are software, the input/output mechanism here is not viewed as message passing, but viewed as method calling. The model can capture the *input assumptions* and the *output guarantees*. The input assumption of a component is the order in which the component's methods are called (assumptions about the environment), and the *output guarantee* is the order in which the component calls external methods. The formalism supports the automatic compatibility check between interface models; the main focus is on determining whether two components can interact in a compatible way taking into account of the constraints imposed by individual interface automata. The authors use an *optimistic* approach to composition, and an *alternative* approach to design refinement. The game-theoretic foundation for the compatibility checking problem is mentioned and treated more carefully in (Passerone et al., 2002).

More precisely, an interface automaton is a simple automaton with a set of states and labeled transitions. A transition from state p to state q labeled by an input $?a$ means the method `a()` of the component is called by the environment. Similarly, when a component moves from state p to state q under an output action $!b$, it means the component calls an outside method `b()`. The internal actions are used to describe the internal computation. Interface automata interact with one another through the *synchronization* of input and output actions while internal actions are interleaved asynchronously.

One of the interface automata features is that not all the input actions are enabled at a state. This reflects the assumptions of a component about the environment which are twofold. First, output actions are always accepted by the environment. Second, the environment will not produce input actions that are not available at the current state.

When two or more components interact, there might be an execution trace leading to a state in which a component can call a non-enabled method of the other. Let the set $Illegal(P, Q)$ denotes such illegal states. The approaches to compatibility check and component refinement are treated under a new *optimistic* view. Traditionally, two components are compatible if they can work together without going to an illegal state under *any* environment. Under the optimistic view, component P and component Q are compatible if there *exists* an environment in which states in $Illegal(P, Q)$ are not reachable. Such environments are called legal environments. Accordingly, there is always a "best" environment which accepts all output of $P \otimes Q$, and generates no inputs to $P \otimes Q$.

There is a linear time algorithm to decide whether two components are compatible. To check for the *compatibility* between two component P and Q, we can simply restrict the states of the product $P \otimes Q$ within the *legal* states. The result of this is the composition $P \parallel Q$. If the composition is non-empty then P and Q are compatible.

Under the optimistic view, the *refinement relation* is used to formalize the relation between abstract and concrete versions of the same component. For example, between an interface specification and its implementation. An implementation of a specification relaxes the input assumptions (accepts more inputs), and restricts the output guarantees (produces fewer outputs). More formally, an interface automaton P refines an interface automaton Q if all input steps of Q can be simulated by P, and all the output steps of P can be simulated by Q. Furthermore, there is a specialized *single-threaded* version of the interface automata which assumes that there is only one active thread of control (only one component active at a time) in a system. This assumption reduces the complexity of compatibility check by pruning out redundant states.

The work of (Passerone et al., 2002) introduces the notion of *interface adaptability* using a game-theoretic framework. Two interfaces are said to be *adaptable* if they can be made *compatible* by communicating through a converter satisfying certain specifications. In other words, the converter makes each component believe that it communicates with the other using its own interface protocol. Under the game-theoretic framework, the synthesis of a converter is interpreted as a game played by components and the specifications. The winning strategy of the game can be used to synthesize a converter. This can be seen as an extension of the results of both (Passerone et al., 1998) and (de Alfaro and Henzinger, 2001). In particular, the game-theoretic interface paradigm of (de Alfaro and Henzinger, 2001) is used not only to check compatibility of interfaces, but also to synthesis interface adaptors. This work extends the approach of (Passerone et al., 1998) by allowing to specify the properties of the converter(adaptor) as an extra automaton. The protocol of interfaces are defined using automata. Note that there is no relationship between the alphabets of the automata. It is a specification automaton that defines the partial order in which the symbols can be presented to consumer, and produced by the converter.

A specification is an automaton whose alphabet is derived from the Cartesian product of the alphabets of the two protocols. Specifications are not concerned with the particular form of protocols being considered, but they make precise what the converter can and cannot do.

Synthesis of the converter can be divided into two steps by an automata-based approach. First, the product machine of the two protocols is composed. The direction of the signals is reversed; the input to the protocols becomes the output of the converter, and vice versa. Note that this composition is also a specification for the converter, since, on both sides the converter must comply with the respective protocols. Next, the product machine is composed with the specification to get the converter. This is to ensure that the converter satisfies both interface protocols and the given specification. Some transitions and states of the converter become illegal as a result of this composition. Hence, the final converter is the product machine with all illegal states/transitions removed.

The problem of protocol conversion discussed above can be re-casted into a more generic game-theoretic framework. Under this framework, synthesizing a converter corresponds to solving a game: Can the converter, by reading output of the producer, provide inputs for the consumer that satisfy both the interface protocols and the specification? The game is played between the protocols and the specification, on one side, and the converter on the other side. In the given example, the goal of the game for the converter, is to ensure that if the producer emits signals according to its definition, then the converter produces signals that corresponds to the transition of the consumer and the specification.

The solution of the above example can be solved by using memory-less strategies which are entirely classical. When the specification includes a fairness condition ϕ (described in temporal logic), the winning strategy must be history-dependent. Games with fairness conditions and history-dependent strategies are examples of games with w-regular wining conditions, and there exist methods to solve them. From the resulting winning strategy, the converter can be easily synthesized as an automaton. In the example given (also in (Passerone et al., 1998)), one component merely produces signals, and the other merely receives signals. In a more realistic setting, a component might both sends and receives signals. The question is whether it is hard to formulate the synthesis converter problem into game-theoretic framework when components both send and receive signals?

3.2 Summary of Previous Work

We now summarize the main features of previous work and attempt to compare with our work.

- All of the previous work only deal with point to point communication, in the other words, there are only two components talking with each other. Even though some of the work (de Alfaro and Henzinger, 2001; Passerone et al., 2002, 1998) can be extended to cope with multiple component situations, it is not clear whether it is scalable to do so. For instance, in (Passerone et al., 2002) to specify the property of the resulting protocol converter, the designer needs to construct a property automaton whose alphabet is the Cartesian product of all the alphabets. This will become unmanageable when the number of parties increase. The work (de Alfaro and Henzinger, 2001) can deal with multiple components, but the focus there is to determine the constraints under which components can interact, no attempt is made to insert a converter to resolve the incompatibilities. To the best of our knowledge, our work is the first one to study the problem of synthesizing protocol converters using scenario-based descriptions which can naturally present the interaction among multiple parties.

- The focus of many of previous works is on a *single* transaction (i.e. roughly corresponding to a single *mode of interaction* in our terms) with all information flowing in one direction (Narayan and Gajski, 1995; Passerone et al., 2002, 1998). In our inter-component description of a system, the interaction among components consists of multiple *modes of interaction.* Each mode is described by MSCs in which control/data signals can flow in both ways. As a result, we can model fairly complex systems as demonstrated in chapter 5.

- Based on the SystemC simulation engine, we can easily simulate the protocol converter along with the component interfaces. The approach in (Narayan and Gajski, 1995) also has this ease of simulation. In addition, we produce the MSC depicting the simulation.

- A nice feature in (Passerone et al., 2002) is that additional behavioral constraints may be imposed on the generated converter using a specification automaton. We also extend our previous work (Roychoudhury, Thiagarajan, Tran, and Zvereva, 2004) to allow this feature. This adds extra expressive power to our modeling capability as we will show in Chapter 5.

In conclusion, the existing work on protocol converter synthesis serve as valuable guideposts to our problem domain. However, our scenario-based formulation and solution of the converter synthesis problem is somewhat orthogonal to these work.

Chapter 4

Technique

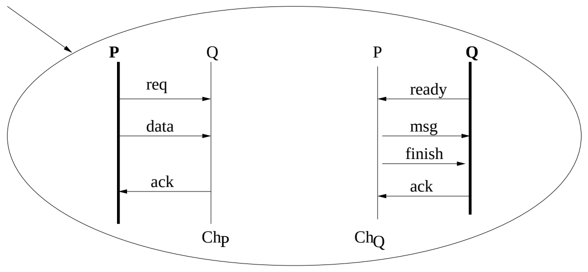


Figure 4.1: A simple HMSC specification of incompatible protocols

k

In this chapter, we outline our technique for automatically generating protocol converters. We first discuss about what the generated protocol converters have to do to bridge the protocol incompatibilities. Next, we present the generated converter as a sequential machine in order to highlight the main features. Our implementation in fact is a multi-threaded one, and this will be brought out in Section 4.2 in which we explain how we generate protocol converters for a mode of interaction - a node in an HMSC. In the section 4.3, we describe how we merge all the converters for individual nodes in an HSMC to get the protocol converters for the whole HMSC. Last but not least, we present how we impose additional constraints on behavior of the generated protocol converters.

The protocols executed by the network of components is assumed to be described as an HMSC, but each node of the HMSC will have a *set* of MSCs associated with it; one MSC for each process taking part in the mode of interaction associated with that node. For convenience, we will refer to this extended notion too as an HMSC. We show a simple example of an HMSC in Figure 4.1. This HMSC describes the interaction between two processes named P and Q. The HMSC has only one node which is, of course, assumed to be the initial node. This node contains two MSCs corresponding to the protocols of P and Q with Ch_P (Ch_Q) describing P's (Q's) view of the protocol. As a matter of fact, Q's (P's) lifeline in Ch_P is simply an assumption made by P (Q)regarding Q'(P's) contribution to the protocol. As long as the events associated with P's lifeline in Ch_P are executed in the order in which they appear, P will walk away from this node with the belief that the protocol associated with this node has been executed correctly.

In the rest of this chapter, we assume that each node n of a HMSC has a set of processes $\mathcal{P}_n$ associated with it, called the *agents* of n. Typically this set is clear from the context. We assume each node n has a family of charts $\{Ch_P\}_{P \in \mathcal{P}_n}$ associated with it; one for each P in $\mathcal{P}_n$. Suppose P is an agent of the node n. By the ***P-view*** at the node n, we mean the sequence of events associated with P's lifeline in the chart Ch_P. Thus, (1) below is the P-view and (2) is the Q-view of the HMSC node shown in Figure 4.1. We note that in the MSC associated with the P-view of a HMSC node, events in the lifelines of processes other than P are not part of the input.

$$\langle P!Q, req\rangle, \langle P!Q, data\rangle, \langle P?Q, ack\rangle \quad (1)$$

$$\langle Q!P, ready\rangle, \langle Q?P, msg\rangle, \langle Q?P, finish\rangle, \langle Q!P, ack\rangle \quad (2)$$

Protocol Analysis The first task in designing (or generating) a protocol converter for incompatible protocols is to understand the functions of the protocols involved. The designer can then decide what functionality can be translated or mapped between incompatible protocols. Many times, communication protocols comprise of different phases/transactions (in our terms, mode of interactions) which implement different functionality. For instance, a communication protocol can have a phase for handshaking, other phase for data transfer, and yet another for error correcting. For that reason, scenario-based descriptions of the incompatible protocols are very useful, since they clearly show how a protocol is structured into phases and where the incompatibilities locate.

In the generating protocol converter problem, we would like to have a converter

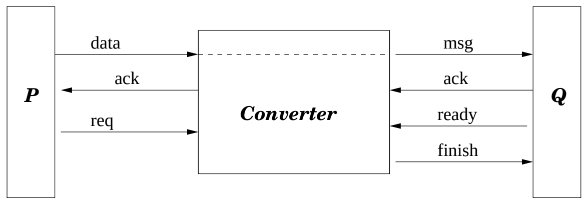


Figure 4.2: Inputs/Outputs of Protocol Converter for Figure 4.1

to make incompatible protocols communicate with one another as if there is no incompatibilities. Because of the fact that messages are basic mechanisms by which protocol functions are implemented, for a protocol converter to be able to mediate the communications between different protocol entities, it must somehow map the messages across components. The scenario-based descriptions can be used, in protocol analysis, to help the designer derive specifications instructing how to translate messages and when to do so.

4.1 Protocol Converters for MSC protocols

We now demonstrate our converter synthesis technique on a single node of the HMSC. Later, we will extend the technique to general HMSCs. Let the agents of the single node be $p_1, \ldots, p_k$ with Ch_{p_i} being denoted for convenience as Ch_i.

First, we partition the whole space of messages(signals) into *control* messages and *data* messages. The difference between them is that data messages carry some content (data) which the converters cannot know before receiving the actual messages. Secondly, both control and data messages can be further classified as follows.

- **Input Messages**: Messages which are sent by a process p_i according to its chart Ch_i, *e.g.* `req`, `data`, `ack` from process Q, and `ready` in Figure 4.1.

- **Output Messages**: Messages which are received by one process p_i according to Ch_i, *e.g.* message `finish`, `ack` received by P, and `msg` in Figure 4.1.

Based on the above categorization, we can identify the inputs and outputs of the desired protocol converter for the specification of Figure 4.1. The *Input* messages will be consumed, and *Output* messages will be produced by the protocol converter.

In our previous work (Roychoudhury et al., 2004), we identify another message category called *shared messages* which are sent by one process p_i according to Ch_i and received by the corresponding process p_j according to Ch_j; the generated converters are allowed to speculative send shared control control messages and relay the shared data messages. However, as noted in (Roychoudhury et al., 2004), the definition of shared messages could lead to name clashes in message names, since more than two components are involved in an HMSC node, in general. Hence, an implicit assumption in our previous work is that message name clashes do not occur (*i.e.* clashes are avoided through renaming if necessary). This also creates difficulties when addressing the issue of data formatting, where the format of data sent by one process may be different from the format of data expected by another process. In such cases, the converter should be able to chop/merge/rotate data packets to satisfy the protocols of both the processes as discussed in (Passerone et al., 1998).

Message relationship specification We tackle this problem by taking in a other input called *message relationship specifications* to assist us on the decision of what messages the converter can automatically generate/consume and what messages it will relay from one side to the other. This can be a result of the protocol analysis, and this approach allows more flexibility than the previous approach we did in (Roychoudhury et al., 2004) as illustrated below.

If M_A and M_B are the message sets of two protocols A and B, then a message relationship specification M_K is a non-empty subset of $\{(m_a, m_b)|m_a \in M_A, m_b \in M_B\}$. Those m_a's and m_b's are called significant messages of A and B. Each (m_a, m_b) consists of exactly one send message and one receive message, and are said to be the image of each other. If message m occurs in M_K more than once, the m is said to be a multi-image message. Typically, M_K specifies what the the data messages must be translated from one side to the other. It also able to described the merging and chopping of data packages in case the data messages are mismatched in sizes.

For instance, for the example, in Figure 4.1, P wants to send to Q a token of data which is the content of the message *data*, however Q expects the data sent as the message *msg*. Therefore, the protocol converter needs to get the message *data* from P and converts it to the message *msg* to send to Q. To instruct the protocol converter to do that we give a message relationship specification $\{(data, msg)\}$. The chopping of data packages can be easily specified too. If a component A would like to send to component B a message a which is 16 bit-wide, but the component B expects two 8 bit-wide messages b and c, the protocol converter needs to get the message a and chop it into b and c. To generate such a protocol converter we will give a message relationship specification $M_K = \{(a, b), (a, c)\}$. The merging of data packages is done in the same way.

Generating the protocol converter We now show the protocol converter generation for one node HMSC specification in Figure 4.1. For simplicity, we generate a protocol converter which automatically sends and receives messages which are not specified in any message relationship specifications. For "important" messages which are specified in some M_K - those need converting and relaying - the protocol converter will first receive all the necessary messages before combining or chopping into new messages to send out. So the message relationship specification M_K we use in this example is $\{(data, msg)\}$. Our protocol converter for the example in Figure 4.1 does the following:

- It first receives the input messages `req` and `ready` which may arrive in any order.
- After `req` is received, the converter waits to receive the data message `data` from P; after `data` arrives, the converter converts it to `msg` and sends to Q.
- At this stage, P is expecting to receive `ack` while Q is expecting to receive message `finish` before sending message `ack`.
- So at the interface with P the converter will generate `ack`, and at the interface with Q the converter will generate output message `finish` before waiting to receive `ack`. The actions at two sides can be interleaved.

The transition system corresponding to the protocol converter described above appears in Figure 4.3. We note that it has a unique *initial state* and *final state*. In fact, this will always be the case for the converter for a single node n of an HMSC. The initial (final) state corresponds to the situation where none (all) of the agents of n have started (completed) the events in their own view of node n. If the agents of the node n are $p_1, \ldots, p_k$, then the sequence of events σ_i in the p_i-view of n induces a sequence of states of length $\mid \sigma_i \mid$ whose prefixes keep track of which events of σ_i have happened so far. If this sequence is $s_{i,1,}, \ldots, s_{i,n_i}$ then: (a) the states of the converter's transition system are drawn from $\{s_{1,1}, \ldots, s_{1,n_1}\} \times \ldots \times \{s_{k,1}, \ldots, s_{k,n_k}\}$, (b) the initial state of the converter is $s_{1,1} \times \ldots \times s_{k,1}$ and (c) the final state of the converter is $s_{1,n_1} \times \ldots \times s_{k,n_k}$.

We now present more formal descriptions; these are based on the work of (Tran, 2004; Zvereva, 2004). We start with the formal definition of communicating finite state machines.

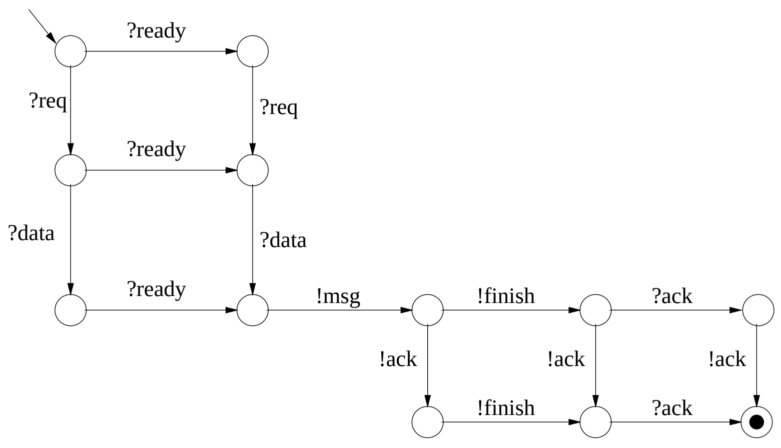


Figure 4.3: Protocol converter for the specification of Figure 4.1

Definition 2. A quintuple $P = (S, q_0, F, M, \delta)$ is a *communicating finite state machine (cfsm)* where:

- S is a finite set of *states*.
- $q_0 \in S$ is the *initial state*.
- $F \subseteq S$ is the set of *final states*.
- M is a finite set of *messages*, $M =!M \cup ?M$. $!M$ and $?M$ are not necessarily two mutually disjoint sets. An element m of M is an *input* if $m \in\ ?M$, and an *output* if $m \in\ !M$.
- δ, the *transition relation*, is a subset of $S \times M \times S$.

A MSC specifying a protocol of component P has several life-lines, one for each of the participating components. However, we are only interested in the life-line of component P, since the actions located along this life-line show the order in which P is supposed to send and receive messages. We can view the life-line of component P as a simple *cfsm* which starts from the top of the life-line and executes the send/receive actions in sequence. Figure 4.4 shows the corresponding *cfsm* of the Q component in the example in Figure 4.1.

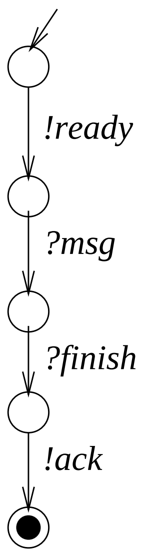


Figure 4.4: Communicating finite state machine.

Denote M_P as the set of messages along the life-line of component P in an MSC. By the definition of MSCs, M_P is an ordered set $\{m_1, m_2, \ldots, m_n\}$ in which $m_i \preceq m_j$ if $i < j$ for $i, j \in \{1 \ldots n\}$. From an MSC specifying the view of a component P, we can formally construct corresponding *cfsm* as $(S_P, q_0^P, F_P, M_P, \delta_P)$. In particular,

- $S_P = \{s_0, s_1, \ldots, s_n\}$ is set of the control locations inserted along the life-line,
- $q_0^P = s_0$,
- $F_P = \{s_n\}$,
- M_P is the set of messages along the life-line described above,
- $\delta_P = \{(s_i, m_{i+1}, s_{i+1}) \mid \text{ for all } i = 0 \ldots n-1\}$.

That is we have a total order for both control locations and the messages. As usual, we assume finite state machines communicate with each others through FIFO channels. So between two component P and Q, there is a FIFO channel from P to Q and another channel from Q to P. However, when we insert between them a protocol converter; P will only communicate through FIFOs with the converter but not with Q; similar situation holds for Q.

Definition 3. Given two *cfsms* which represent protocols component P and Q, a set M_K is a *message relationship specification* which satisfies

- $M_K \neq \emptyset$,
- $M_K \subset \{(m_p, m_q) | m_p \in M_P, m_q \in M_Q\}$,
- For every $(m_p, m_q) \in M_K$ either $m_p \in !M_P \wedge m_q \in ?M_Q$ or $m_p \in ?M_P \wedge m_q \in !M_Q$.

The message relationship specification is a non-empty set of message pairs. A pair in the set must include an input message of one component and an output message of other component.

Definition 4. Given two *cfsms* which represent protocols component P and Q with a message relationship specification M_K we can construct the interface as a *cfsm* $H = (S_H, q_0^H, F^H, M_H, \delta_H)$ which consists of the following elements:

- $S_H = S_P \times S_Q$,
- $q_0^H = (q_0^P, q_0^Q)$,
- $F^H = \{(q_n^P, q_m^Q)\}$ where $q_n^P \in F_P$ and $q_m^Q \in F_Q$,
- $M_H = M_P \cup M_Q$, $!M_H = ?M_P \cup ?M_Q$ and $?M_H = !M_P \cup !M_Q$,

- $$\begin{aligned}
\delta_H \;=\; & \{((p,q),m,(p\prime,q)) \mid (p,m,p\prime) \in \delta_P \\
& \wedge \text{ there not exists } (m,m_q) \in M_K \text{ for some } m_q\} \\
\bigcup \; & \{((p,q),m_p,(p\prime,q)) \mid (p,m_p,p\prime) \in \delta_P \\
& \wedge \text{ there exists } (m_p,m_q) \in M_K \text{ for some } m_q \text{ and } m_p \in !M_P\} \\
\bigcup \; & \{((p,q),m_q,(p,q\prime)) \mid (q,m_q,q\prime) \in \delta_Q \\
& \wedge \text{ there exists } (m_p,m_q) \in M_K \text{ for some } m_p \text{ and } m_q \in !M_Q\} \\
\bigcup \; & \{((p,q),m,(p,q\prime)) \mid (q,m,q\prime) \in \delta_Q \\
& \wedge \text{ there not exists } (m_p,m) \in M_K \text{ for some } m_p\} \\
\bigcup \; & \{((p,q),m_p,(p\prime,q)) \mid (p,m_p,p\prime) \in \delta_P \\
& \wedge \forall (m_p,m_q) \in M_K \text{ where } m_p \in ?M_P \text{ then } m_q \preceq_Q q\} \\
\bigcup \; & \{((p,q),m_q,(p,q\prime)) \mid (q,m_q,q\prime) \in \delta_Q \\
& \wedge \forall (m_p,m_q) \in M_K \text{ where } m_q \in ?M_Q \text{ then } m_p \preceq_P p\}
\end{aligned}$$

In short, given two protocols and a specification about what messages needed to be converted between them, we can construct a protocol converter as a product state machine of the two protocols. The starting state of the machine is a state where two protocol components are at their starting states; similarly to the final state. The outgoing messages of the protocols now become the incoming messages of the protocol converter; similarly the incoming messages of the protocols now become outgoing messages of the protocol converter. For the messages not specified in the message relationship specification, the protocol converter can just send to (receive from) components. For the incoming messages (as outgoing for the components) which are specified in the message relationship specification, the protocol converter will take in. The protocol converter actually remember those messages; so it should have some form of memories; however at this level of abstraction we do not express this information. The protocol converter will send out those outgoing messages specified in the message relationship specification when the component want to receive and when the protocol converter itself has received the corresponding message (it is a pair) from the other component.

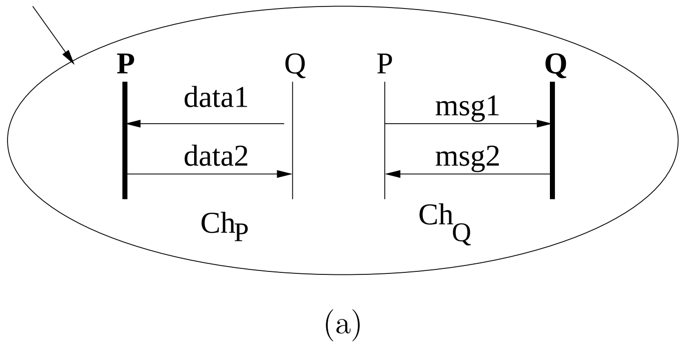
Incompatibility Detection As mentioned earlier, the converter cannot speculatively send important messages - those specified in some message relation specifications - which are usually data messages. Therefore in certain situations we cannot synthesis a protocol converter; for example, the Figure 4.5(a) shows an such example. In this situation, component P expects to receive message $data_1$ before sending message $data_2$ while component Q expects to receive message msg_1 before sending message msg_2. By analyzing the two protocol semantics, we find out that $data_1$ corresponds to msg_2 and msg_1 corresponds to $data_2$. Hence, a protocol converter needs to receive msg_2 from Q and to convert it to $data_1$ to send to P; the converter also needs to receive $data_2$ from P and to convert it to msg_1 to send to Q. The message relationship specification describing this requirement is $M_K = \{(data_1, msg_2), (msg_1, data_2)\}$.

However, no protocol converters can handle this situation. Because the first thing a protocol converter has to do here is to either send $data_1$ to P or send msg_1 to Q which the converter cannot do since it does not have the content. In the other words, we have a deadlock situation. Consequently, in the process of synthesizing a protocol converter as a finite state machine, there will be a state in which the only moves available are speculatively sending "important" messages; in this case it is the starting state. Hence we declare incompatibility and abort. We give the formal definition as follows.

Definition 5. We say that protocols of component P and Q are *adaptable*, if and only if, the final states F^H of the protocol converter are *reachable* from the initial state.

If two protocols are not *adaptable*, we say they are *incompatible*.

From the protocol converter synthesis angle, when the two protocols are not



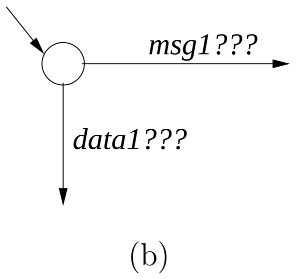


Figure 4.5: (a) A pair incompatible protocols with $M_K = \{(data_1, msg_2), (msg_1, data_2)\}$ (b) A converter for the specification in (a)

compatible, we cannot synthesis a functional protocol converter. That means the protocol converter synthesized according to the Definition 4 when put in between two protocols will create dead-lock situations.

4.2 Multi-threaded Protocol Converters

In the previous section, we have built the converter as a single sequential finite state transition system. In general, for deriving a converter involving k processes $p_1, \ldots, p_k$ (described via k MSCs) we view the converter as a multi-threaded program with k threads $T_1, \ldots, T_k$. Each thread T_i of the converter communicates with exactly one process p_i. Any interleaving of events across converter threads is allowed. Any two converter threads communicate with each other via point-to-point message passing. In particular, for a converter with k threads we will have $k(k-1)$ message buffers; buffer $q_{i,j}$ contains messages sent by converter thread T_i to converter thread T_j. This model of the converter is a more faithful reflection of SystemC protocol converters automatically generated by our toolkit.

Why do the converter threads need to communicate? Recall that converters have to match and convert messages specified in message relationship specifications.

For instance, in the example in Figure 4.1 message *data* from P is matched with message *msg* to Q, hence the converter thread servicing process Q needs to know if the converter thread servicing process P has received the message *data* so that it can convert *data* to *msg* to send to Q. Clearly only "important" messages will be exchanged between converter threads via message buffers. Other than that any converter threads consumes input messages and generates output messages without communicating with other threads.

We develop a multi-threaded protocol converter for the HMSC specification of Figure 4.1 as follows. The converter has two threads T_P (communicating with P) and T_Q (communicating with Q). The sequence of events executed by T_P and T_Q are obtained from the P-view and Q-view in Figure 4.1. According to the P-view, T_P executes the sequence of events $?\mathtt{req}, ?\mathtt{data}, !\mathtt{ack}$ and T_Q executes the sequence of events $?\mathtt{ready}, !\mathtt{msg}, !\mathtt{finish}, ?\mathtt{ack}$. Any interleaving of T_P and T_Q is allowed. The task involved within the converter in the events mentioned in the preceding are different. Since `req` and `ready`, and `ack` from Q are input messages, their receipt only requires the converter to wait until the messages actually arrive. The message `finish` and `ack` to P are output messages, so they are simply sent off by corresponding converter threads. On the other hand, `data` needs to be converted to `msg`, therefore when T_P executes `?data` it appends `data` to the buffer $q_{P,Q}$. And when T_Q executes `!msg` it first checks whether `data` is in $q_{P,Q}$; if so, it removes `data` from $q_{P,Q}$, converts it to `msg`, and sends off to process Q. Figure 4.6 describes the multi-threaded implementation of our protocol converter.

We now formally define the definition of the protocol converter for more general cases in which multiple components interact with each others.

Definition 6. Denote $\mathcal{P}$ is the set of N components. The *cfsm* $P_i = (S_i, q_0^i, F_i, M_i, \delta_i)$ is extracted from the view(MSC) of components $P_i \in \mathcal{P}$ with $i = 1 \ldots N$. The set $M_K \subseteq \{(m_i, m_j) | m_i \in M_i, m_j \in M_j \, 1 \leq i, j \leq n, i \neq j\}$ is a message relationship

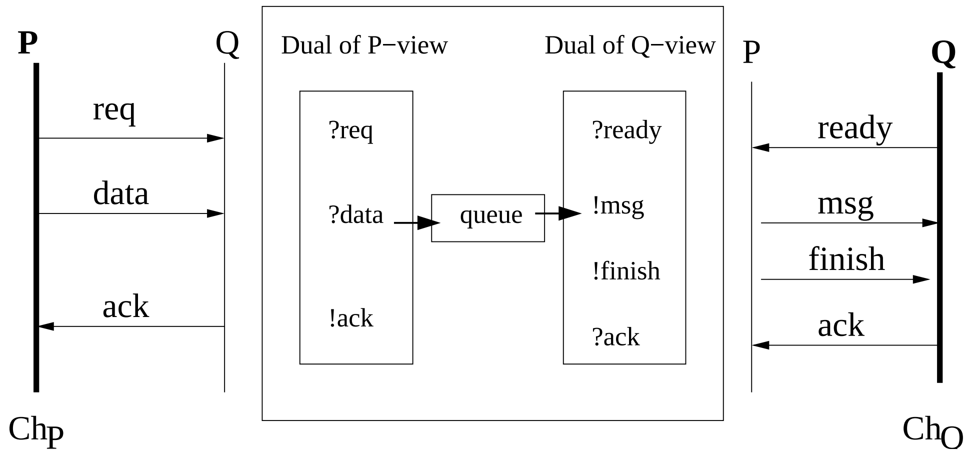


Figure 4.6: The Multi-threaded Protocol Converter for the HMSC in Figure 4.1

specification. The protocol converter will be a *cfsm* $H = (S_H, q_0^H, F^H, M^H, \delta^H)$ in which the elements are:

- $S_H = \prod_{i=1\ldots N} S_i$,
- $q_0^H = (q_0^1, q_0^2, \ldots, q_0^N)$,
- $F^H = \{(q_1, q_2, \ldots, q_N)\}$ where $q_i \in F_i$.
- $M_H = \bigcup_{i=1,\ldots,N} M_i$.
- $$\begin{aligned} \delta_H \;=\;& \{((p_1,\ldots,p_i,\ldots,p_N), m_i, (p_1,\ldots,p_i\prime,\ldots,p_N)) \mid \\ & (p_i, m_i, p_i\prime) \in \delta_i \wedge \text{ there not exists } (m_i, m) \in M_K \text{ or } (m, m_i) \in M_K \text{ for some } m\} \\ \bigcup\;& \{((p_1,\ldots,p_i,\ldots,p_N), m_i, (p_1,\ldots,p_i\prime,\ldots,p_N)) \mid \\ & (p_i, m_i, p_i\prime) \in \delta_i \wedge \text{ there exists } (m_i, m) \in M_K \text{ or } (m, m_i) \in M_K \text{ for some } m \\ & \wedge m \in !M_i\} \\ \bigcup\;& \{((p_1,\ldots,p_i,\ldots,p_N), m_i, (p_1,\ldots,p_i\prime,\ldots,p_N)) \mid \\ & (p_i, m_i, p_i\prime) \in \delta_i \wedge \text{ there exists } (m_i, m_j) \text{ or } (m_j, m_i) \in M_K \text{ for some } m \\ & \wedge m_i \in ?M_i \wedge\; m_j \preceq p_j \text{ for all such } m_j\} \end{aligned}$$

A implicit assumption here is that for each HMSC node we will have only one message relationship specification, but it is not a limitation. Theorectially, we can easily combine/decompose message relationship specifications.

4.3 Protocol Converter for HMSC-specifications

We now extend our converter generation technique to arbitrary HMSC specifications. For this purpose, we need to deal with (a) concatenation of nodes (b) branching behavior (*i.e.* a node in the HMSC having multiple immediate successors) and (c) loops.

Since we adopt asynchronous concatenation, it is appropriate to view the converter at each node as a multi-threaded system (refer Section 4.2). If we have a protocol involving k processes and the converter threads are $T_1, \ldots, T_k$, the converter for a sequence of nodes $n_1, \ldots, n_N$ can be synthesized by connecting the end of thread T_j (for all $1 \leq j \leq k$) in node n_i to the beginning of thread T_j in node n_{i+1}. Note that even for synchronous concatenation, we can develop a multi-threaded converter for each node of the HMSC; however the converter threads will synchronize at the end of each HMSC node.

We now consider branching. If a node n of an HMSC has multiple successors (say $n \rightarrow n_1$ and $n \rightarrow n_2$), we need to ensure that *all* the converter threads move to either n_1 or n_2 (*i.e.* we want to prevent the situation where certain threads move to n_1 and the others move to n_2 as this will generate behaviors not allowed by the HMSC specification (Uchitel, Kramer, and Magee, 2001)). Since we are working with asynchronous concatenation, several nodes of the HMSC may be active at any point of time ; we require an external thread (which we call the *environment thread*) to decide on the immediate successors of each of these nodes. This decision is assumed to be available in a channel called *RUN*. Each agent of a node and each converter thread associated with the node, on completion, will query the *RUN* channel to determine which node it should enter (or which thread it should pass control to) next. In our actual implementation, the *environment thread* is a simple file containing a finite run of the HMSC - a path in the HMSC starting from the initial node. However, it is easy to extend the implementation such that the environment thread is modeled as a non-deterministic process.

Finally, we deal with loops. For HMSCs containing loops, a process can unboundedly overtake another since asynchronous concatenation of nodes is assumed. This will create unboundedly many active nodes of the HMSC. To avoid this situation, we do not allow multiple active copies of the same node at any stage in any execution. More precisely, we allow only cycle-bounded executions (as defined in Chapter 2). This policy is easily enforced by the *RUN* channel. Along the path supplied to it (either statically as in our current implementation or dynamically) by the *environment* thread, it keeps track of where each process is. It also keeps track of the currently active nodes. Thus, it provides the name of the next node (say n') to any process p exiting a node of the HMSC only if the node n' is not currently active. If the next node is still active, it provides a *WAIT* signal as a response which blocks p from proceeding. Naturally, this policy is also applied to the individual threads of the converter.

We provide a formal treatment by starting with the definition of our HMSC.

Definition 7. An HMSC is a tuple $G = (\mathcal{T}, \tau_0, \mathcal{F}, \mathcal{E})$ where:

- $\mathcal{T}$ is a finite set of transactions. Each transaction τ is of the form $\{[Ch_i]\}_{i=1}^{N}$ where Ch_i is a MSC presenting the view of component P_i.
- $\tau_0 \in \mathcal{T}$ is the initial transaction.
- $\mathcal{F} \subseteq \mathcal{T}$ is set of final transactions.

- $\mathcal{E} \subseteq \mathcal{T} \times \mathcal{T}$ is the set of edges.

The HMSCs we are defining here are similar to the HMSC definition in Chapter 2. Except that in each node there is a set of MSCs; each MSC represents the view(protocol) of a component in that node. We now describe the protocol converter synthesizing for inter-component HMSC descriptions. First, we need to extract a *cfsm* for each component in the HMSC, then we synthesize the converter from there. Previously, we have already defined the *cfsm* which presents the view of each component in a single node. The *cfsm* of component P in the HMSC will be a concatenation of *cfsms* of P in all the nodes according to the edges. Let call $P_\tau = (S_\tau, q_0^\tau, F_\tau, M_\tau, \delta_\tau)$ be the *cfsm* of component P in the node τ. Notice that we define the *cfsms* below to be used to synthesize the converter. They are not the equivalent representations of the components of the HMSC.

Definition 8. Given an HMSC, the *cfsm* of representing the component P is $(S_P, q_0^P, F_P, M_P, \delta_P)$ where

- $S_P = \bigcup_{\tau \in \mathcal{T}} S_\tau$,
- $q_0^P = q_0^{\tau_0}$,
- $F_P = \bigcup \{F_\tau \mid \tau \in \mathcal{F}\}$,
- $M_P = \bigcup \{M_\tau \mid \tau \in \mathcal{F}\}$, $!M_P = \bigcup \{!M_\tau \mid \tau \in \mathcal{F}\}$ and $?M_P = \bigcup \{?M_\tau \mid \tau \in \mathcal{F}\}$
- $$\begin{array}{rcl} \delta_P & = & \{(p, m, q) \mid (p, m, q) \in \delta_\tau \text{ where } \tau \in \mathcal{T}\} \\ & \bigcup & \{(p, \epsilon, q) \mid p \in \mathcal{F}_\tau \wedge q = q_0^{\tau\prime} \wedge (\tau, \tau\prime) \in \mathcal{E}\} \end{array}.$$

In the definition above there are special transitions marked with ϵ. They are points where components finish their work in one HMSC node and move to the next one. As mentioned earlier, the semantics of HMSCs assume all the components make consistent choices. In our implementation, we use an environment thread to decide. With this semantical assumption, the protocol converter defined in Definition 6 can now be extended for HMSCs in the same manner.

4.4 Additional Behavioral Requirements

So far, our protocol converters will automatically generate and consume messages (signals) when needed and will convert important messages according to given message relationship specifications to allow incompatible components communicate with each other while still using their own native protocols. Our technique is able to deal with fairly complicated systems. However, in many situations we need to impose additional behavioral requirements on generated protocol converters. We would like to have a mechanism to specify additional requirements on the behavior of a generated protocol converter on top of what it has done. For example, we would like to be able to say things like certain messages cannot be sent until some conditions are satisfied, or certain sequences of actions are not allowed.

One reason for such a mechanism is that control messages cannot always speculatively generated because they might represent access authorization to shared resources which are not explicitly modeled in an HMSC. So in these kind of situations, we would like to control the send and receive of certain messages so that only one component can access the shared resource at any moments. Another reason for which we need additional behavioral requirements is to make generated protocol converters to have some desired properties. For example, let us look at the example in Figure 4.7, to be able to make the *sender* and the *receiver* communicate to each other we only need to specify a message relationship specification to be $M_K = \{(msg, data)\}$. However, if we would like to limit the buffer size of the converter such that only one message is in transit at any moment, we possibly want to impose that the `ack` message is sent only after the `data` message is delivered to the *receiver*. Or if the *sender* runs faster than the *receiver* and it cannot tolerate any delay, we can impose that the `ack` message is sent immediately after `msg` is received - assume that sending `data` takes a certain time. In this specific example, the additional behavioral requirements, which we want to impose, only involve actions in one node of the HMSC, but, in general, we would like to have a mechanism to impose additional requirements on overall behavior of protocol converters; in other words, an additional behavioral requirement might relate messages across the whole HMSC.

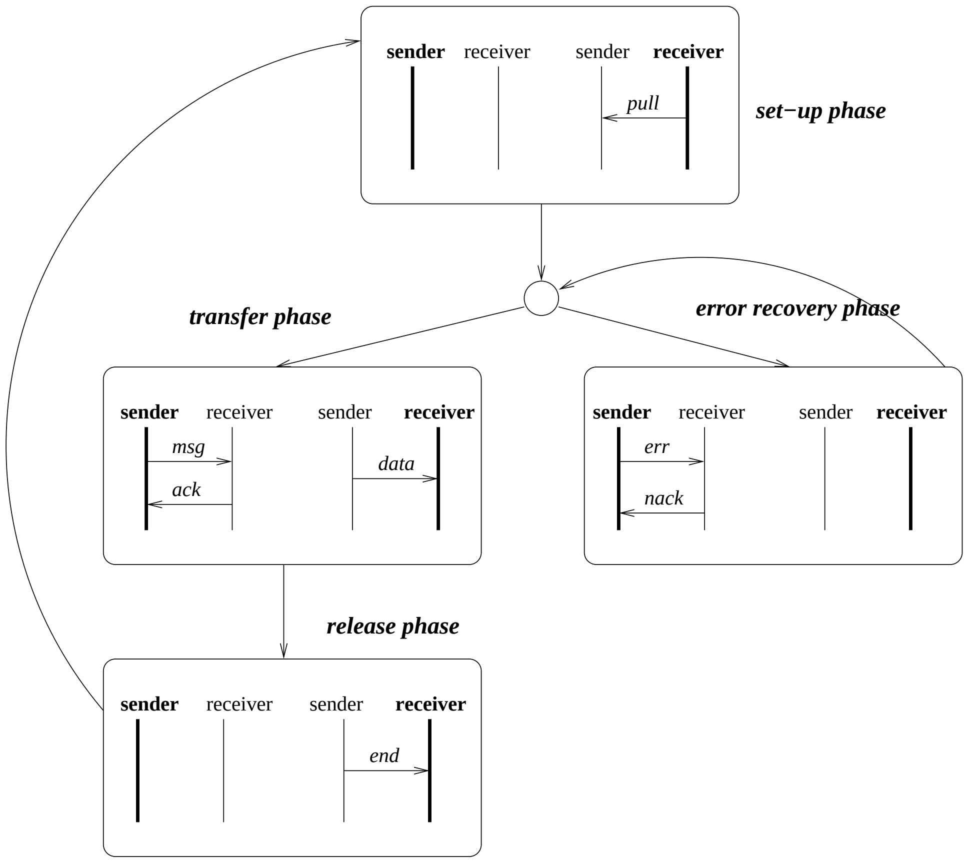


Figure 4.7: An HMSC example

Behavioral Specifications We use automata to specify additional behavioral requirements; we can call them behavioral specifications. A behavioral specification S_K is a finite state machine (S, s_0, M, δ) which only has one starting state and all states are ending states; the alphabet is a subset of all messages of all components $M \subseteq \bigcup_{P\in\mathcal{P}} M_P$. As stated a S_K specifies some desired properties of the whole system. For example, a protocol converter may be designed to provide - besides bridging the incompatibilities - end-to-end or local acknowledgment depending on how S_K is specified.

Definition 9. A behavioral specification is a tuple (S, s_0, M, δ) where

- S is the set of states,
- $s_0 \in S$ is the initial state,
- $M \subseteq \bigcup_{P\in\mathcal{P}} M_P$ is the set of special messages,
- δ is the transition function.

Formally, a S_K restricts what are legal behavior of a generated protocol converter; more precisely, given a run σ of a protocol converter, we say σ is a *legal* run if $\sigma_{|M} \in \mathcal{L}(S_K)$. In other words, any sequences of actions of a protocol converter after removed all the unrelated messages must be accepted by S_K. For a scenario-based description, we can have more than one behavioral specifications, each describes some desired properties of the protocol converter. The behavioral specification which limits the buffer size of the converter for the example in Figure 4.7 is shown in the Figure 4.8.

For a given behavioral specification, we implement it as a monitor to the protocol converter generated using techniques described in above sections. More precisely, given a protocol converter and a behavioral specification S_K, we will monitor all the actions sending or receiving a message which is in the alphabet M. If an

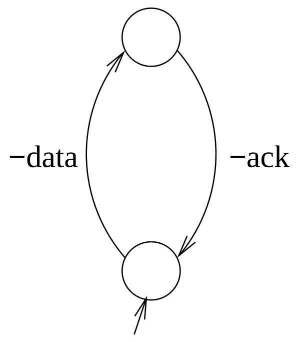


Figure 4.8: The behavioral specification which limits the converter buffer size of the example in Figure 4.7

action - send or receive message m - is ready to execute, the converter will check with the corresponding monitor to see if there is a transition labeled m available at the current state of S_K. If there is, the converter will go ahead to execute the action and inform the monitor to update the current state of S_K; otherwise, the action is blocked until there is a transition labeled m available at S_K. Note that the converter as a whole is not blocked; recall that our protocol converters are implemented as multi-thread programs. So every action belongs to some threads; if an ready action in thread T_i is blocked due to an behavioral specification, actions in other threads are still free to go. In the next chapter, we will show more examples to illustrate the use of behavioral specifications.

Definition 10. Given a protocol converter $(S_{\mathcal{C}}, q_{\mathcal{C}}, M_{\mathcal{C}}, \delta_{\mathcal{C}})$ and a behavioral specification $(S_{\mathcal{K}}, s_0, M_{\mathcal{K}}, \delta_{\mathcal{K}})$ where $M_{\mathcal{K}} \subseteq M_{\mathcal{C}}$, the converter which comply to the behavioral specification is a cfsm (S, q, M, δ) where

- $S = S_{\mathcal{C}} \times S_{\mathcal{K}}$,
- $q = (q_{\mathcal{C}}, s_0)$,
- $M = M_{\mathcal{C}}$,

- $\delta = \{((p,q), m, (p\prime, q)) | (p, m, p\prime) \in \delta_{\mathcal{C}} \wedge m \notin M_{\mathcal{K}}\}$
  $\bigcup \{((p,q), m, (p\prime, q\prime)) | (p, m, p\prime) \in \delta_{\mathcal{C}} \wedge (q, m, q\prime) \in \delta_{\mathcal{K}}\}$

As described we use message relationship specifications to guide the converter regarding message conversion. Theoretically speaking, message relationship specifications can be, however, described by behavioral specifications. For example, the $M_K = \{msg, data\}$ is actually an syntactical sugar form of the automaton in Figure 4.9. This automata imposes requirements on the message sequence of legal runs, but it does not explicitly specify the message relationship. Therefore, for the practical purposes we prefer to use sets of message tuples to represent message relationship specifications. Nevertheless, by using automata to represent all kind of requirements a designer would like to impose/guide the process of generating protocol converters, we achieve a unified framework to reason about the system.

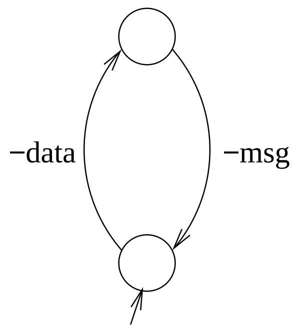


Figure 4.9: Automaton representation of $M_K = \{msg, data\}$

Chapter 5

Applications and Case Studies

In this chapter, we use selected features of some SoC bus communication protocols to illustrate the use of our converter synthesis technique. We mainly model the protocols of the bus master(s) and the bus slave. In other words, each node of the input HMSC specification contains a MSC for each master and each slave. However, the bus masters can communicate with the bus slaves only with the help of the bus controller. We have modeled several features from existing SoC bus protocols which enable high speed data transfer *e.g.* split transactions, different bus priorities, concurrent transactions, etc. Many of these features are common in SoC bus protocols such as AMBA AMBA, CoreConnect Architecture etc. We will use different examples to incrementally show various capabilities of our approaches.

5.1 Basic Bus Communication Protocols

In the following, we do not model the bus-controller as a separate process. Instead, we synthesize the protocol converter which enables communication between masters and slaves. Note that in the next two examples we do not impose any constraints on the synthesized protocol converter beyond enabling masters/slaves to execute their individual projected views. The work of (Passerone et al., 2002) allows specification of additional temporal properties of the converter to be synthesized via a "specification automaton". Our approach also allows this feature by using the behavioral specifications as will being shown in the next section.

The first example is shown in Figure 5.1. It exhibits "split transactions"; thus the *master* which has been granted access to the bus may not get its request serviced because the *slave* cannot service it currently (this is communicated by the slave via a `split` signal). Subsequently, the *master*'s right to access the bus will be ignored (*i.e.* it will not even be considered for bus contention) until the *slave* indicates its willingness to serve the *master* via a `resume` signal. The signal `transfer` shown in Figure 5.1 indicates data transfer (*i.e.* it is a data signal). In this case, there are no incompatibilities in the data transfer; both the *master* and the *slave* agree that when they want to transfer data they will send/receive the `transfer` signal(message). So the message relationship specification is $M_K = \{\texttt{transfer}_{master}, \texttt{transfer}_{slave}\}$. Note though only one master has been modeled, the effect of other masters is implicitly captured via the `grant` and `nogrant` signals.

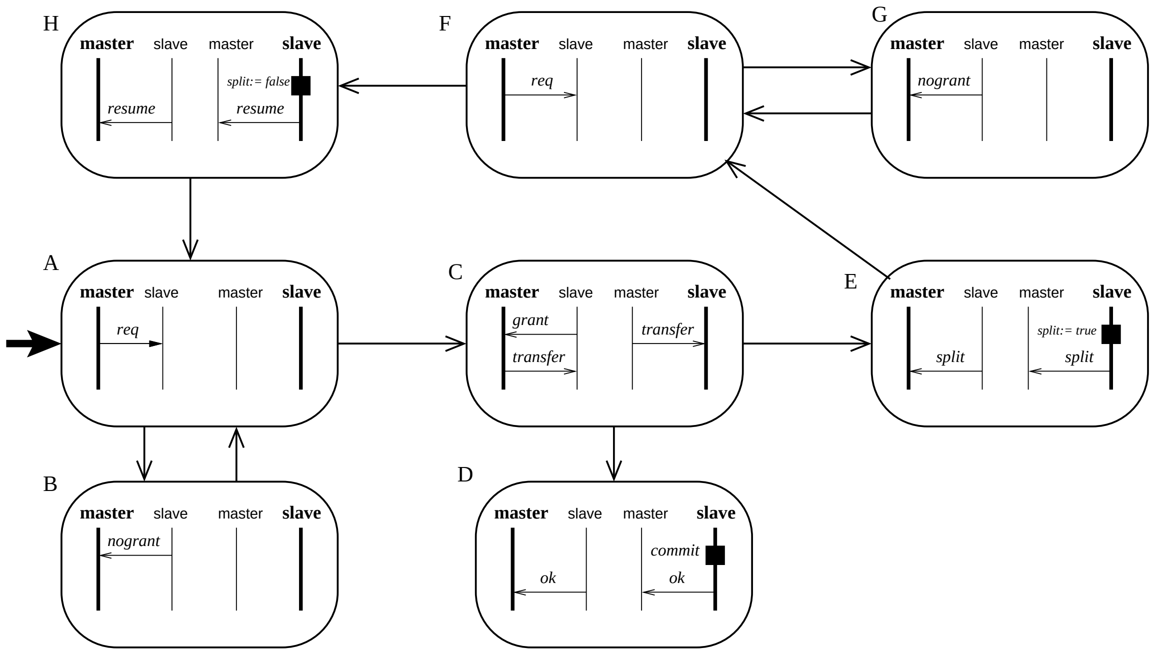


Figure 5.1: Split transfers example

Another example modeling two masters with different priorities (assigned statically) is shown at Figure 5.2. It contains three components: two masters and one slave; the master m_1 has higher priority than the master m_2 in terms of bus access. The bus controller mediates their access and it is synthesized as the converter. Note that m_2 can successfully transmit data (node E) only if m_1 has sent a req_0 message (indicating that it does not want to request bus access); the req_1 signal in this example indicates willingness to access the bus. We have also generated

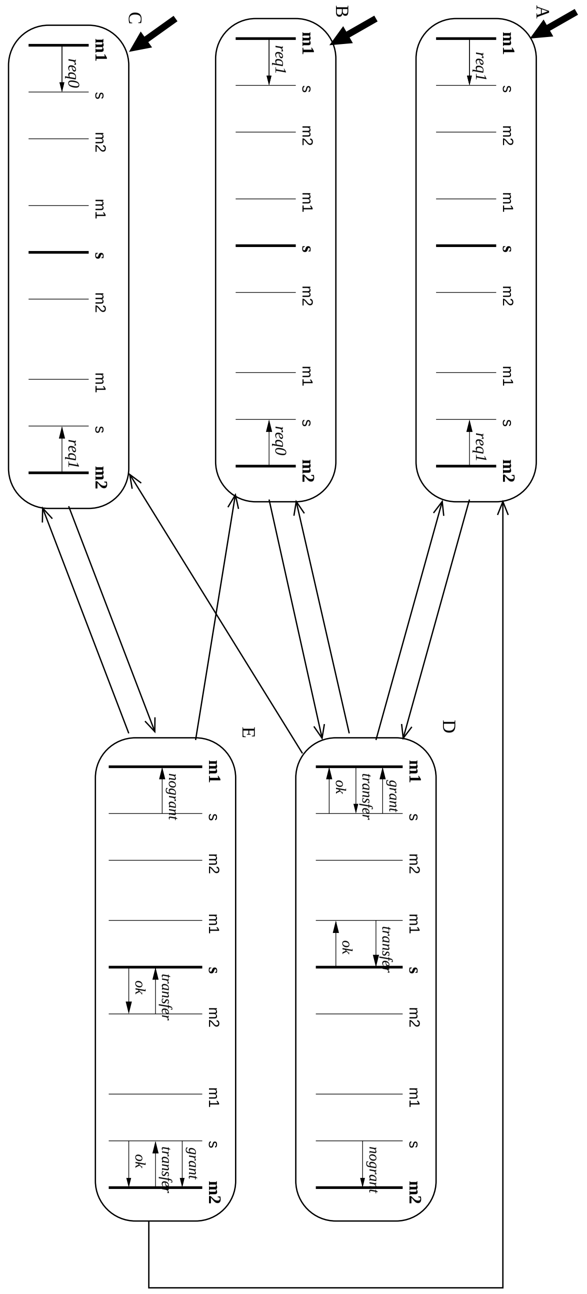


Figure 5.2: Bus transfers guided by master priorities example

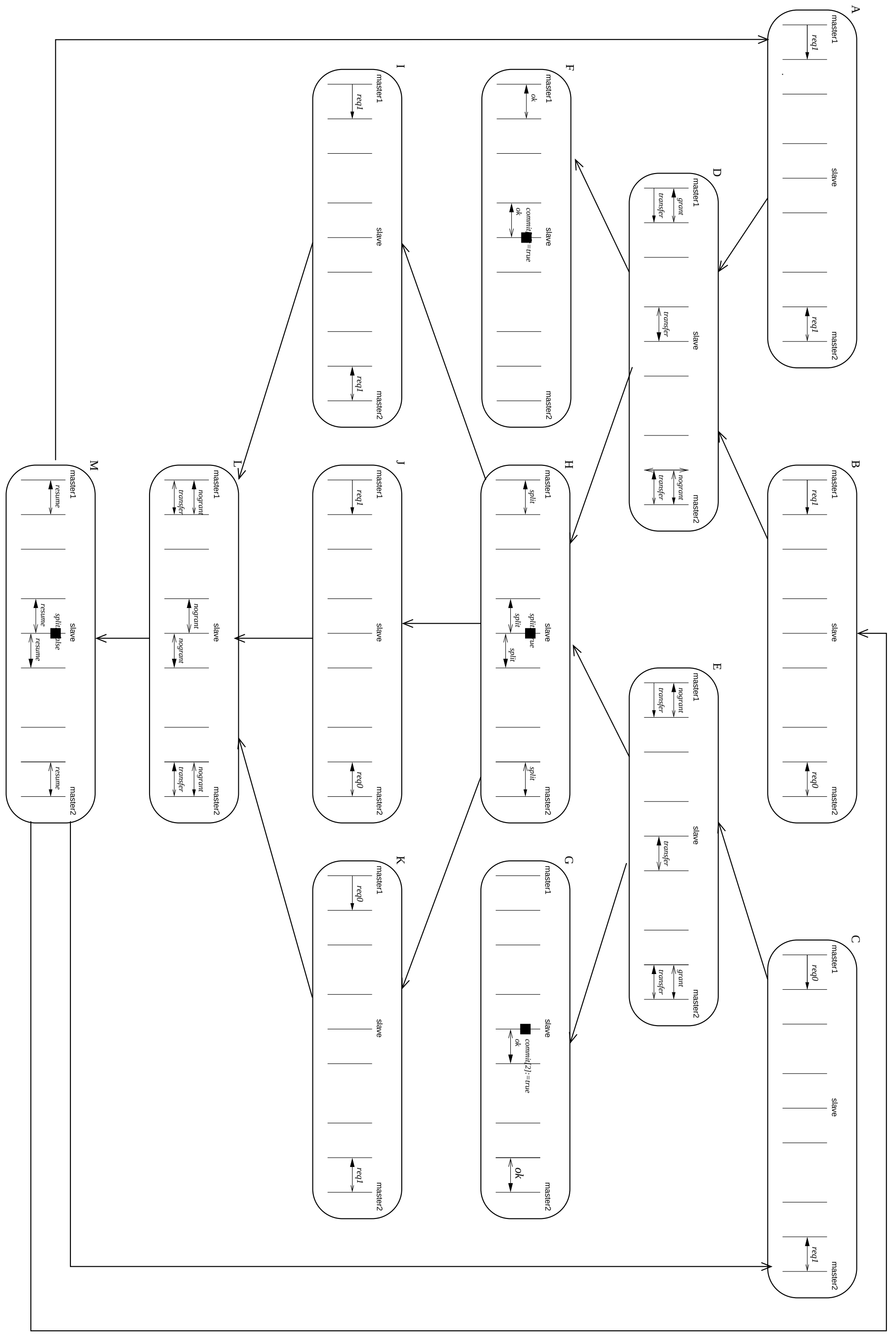
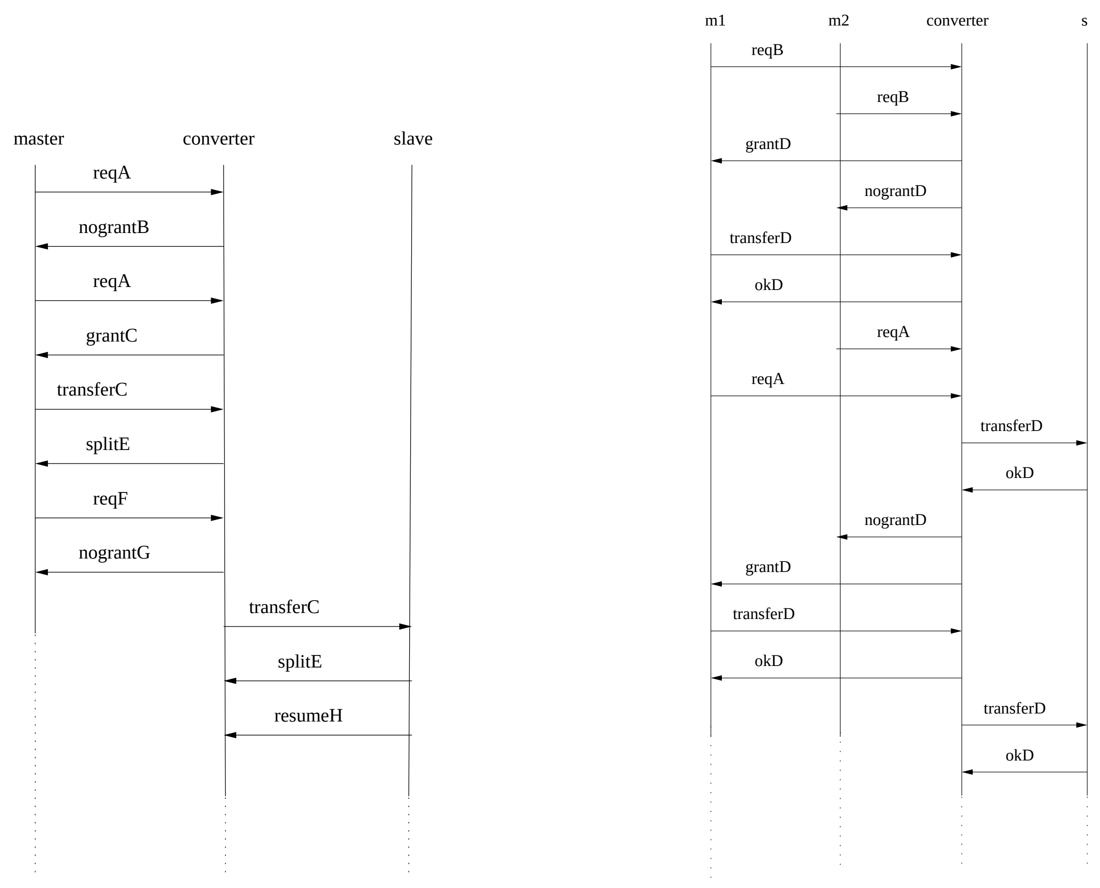


Figure 5.3: Bus transfers guided by master priorities with split transactions example



(a) (b)

Figure 5.4: Simulation run of synthesized converter for (a) Figure 5.1 with input path `ABACEFGFHABACD` and (b) Figure 5.2 with input path `BDADCEAD`

the converter for more involved features of common bus protocols. For example, we have used our converter generator toolkit to synthesize the bus controller of a simple bus protocol involving (i) multiple masters with individual static priorities and (ii) split transactions between master and slave; in other words, the example is the combination of the two above examples; it is shown at Figure 5.3.

Experimental Results Finally, we present the simulation results of the generated protocol converters for the two simple examples we presented in this section.

Figure 5.4(a) shows the simulation run of the synthesized converter for the HMSC specification in Figure 5.1; the input path through the HMSC which was used to drive this simulation is `ABACEFGFHABACD`. This corresponds to the following sequence of scenarios:

- The master requests bus access and is denied (sequence `AB`).
- The master requests bus access again and is granted (sequence `AC`).
- The slave splits master's transfer as a result of which master is ignored as a candidate for bus access (sequence `EFGF`).
- The slave indicates that it is ready to resume communication with master (node `H`).
- The master makes a fresh request for bus access which is considered for contention, but is not granted (sequence `AB`).
- The master makes another request which is granted (sequence `AC`)
- The master's transfer is finally completed (node `D`).

Similarly, Figure 5.4(b) shows the simulation run of the synthesized converter for the HMSC specification in Figure 5.2; the input path through the HMSC which was used to drive this simulation is `BDADCEAD`.

We present simulation runs in a MSC format; this allows us to visualize the interaction between the synthesized converter and the master/slave processes. Each signal shown in the MSCs of Figure 5.4(b) is marked with the node in the HMSC specification to which it belongs; thus the `req` signal of node A is marked as `reqA` and so on. Note that our synthesized converter is responsible for generating the `grant`, `nogrant` signals (*i.e.* it is acting as the bus controller to decide on bus access by masters); at the same time, the converter is also responsible for receiving the request signals from the master(s) since these signals are not visible to slave(s).

5.2 More Advanced Examples

By using message relationship specifications, we are able to handle mismatches in data message names and sizes; and the behavioral specifications allow us to describe additional temporal properties of the converter to be synthesized. These brings us enormous modeling power. The next example, shown in Figure 5.5, models two masters with different priorities (prefixed statically) want to read/write data from/to a slave. The $master_2$ has higher priority than the master $master_1$ in terms of bus access. The synthesized converter acts as the bus controller here to mediates the master accesses.

There are few additional functionalities which the converter handle here. First, the two master use word(32-bit) transfers while the slave uses half-word(16-bit) transfers. Therefore, the converter needs to perform dynamic bus sizing to allow components with different data widths to efficiently communicate. In our example, the converter will read the *data* message from a master and break it into D_1 and D_2 messages to write to the slave, vice versa the converter will collect D_1 and D_2 to merge them together and transfer to a master. The converter uses message relation specifications to guide the merge/split messages. For instance, the *mode of interaction* E is attached with a message relation specification $M_K^E = \{(D_1, data), (D_2, data)\}$, similarly for other modes of interaction.

Second, in this example there is an implicit assumption that the common bus shared by three components - two masters and a slave - has separate read/write wires. That means when both two masters want to read/write to the slave the converter has to mediate, and only one master is granted to access the bus. However, if a master requests to read and the other requests to write, both operations can be overlapped at the same cycle (but on different clock edges) yielding a maximum bus utilization of two data transfer per clock. The scenarios G and H describe these read/write overlapped situations. Note that in order to have two operations in the same cycle, we have to make sure they are done overlapped each other to avoid the bus contention. For instance, in the scenario G, the slave must receive the $start_W$ signal only after the *done* signal is sent to $master_2$; this is to make sure that the bus is free for the $master_1$ to do a write operation. We impose this condition by specifying a behavioral specification as in Figure 5.6.

Another example is shown in Figure 5.8. This example has the same setup with the previous one: two masters with different priorities and one slave using a common bus. In this example, the $master_1$ having higher priority can interrupt a transaction in process of the $master_0$ as shown in the scenario F in Figure 5.8. Only after the *split transaction* is done, the transaction in process of the lower priority $master_0$ is resumed. Beside merely making sure all the component executes the transaction with their own protocol, additional temporal order of message in this scenario is needed. In this situation, the *split* signal only can be sent to $master_0$ after receiving a *req* signal, and following by sending *grant* signal back to $master_1$. This requirement is specified in Figure 5.7.

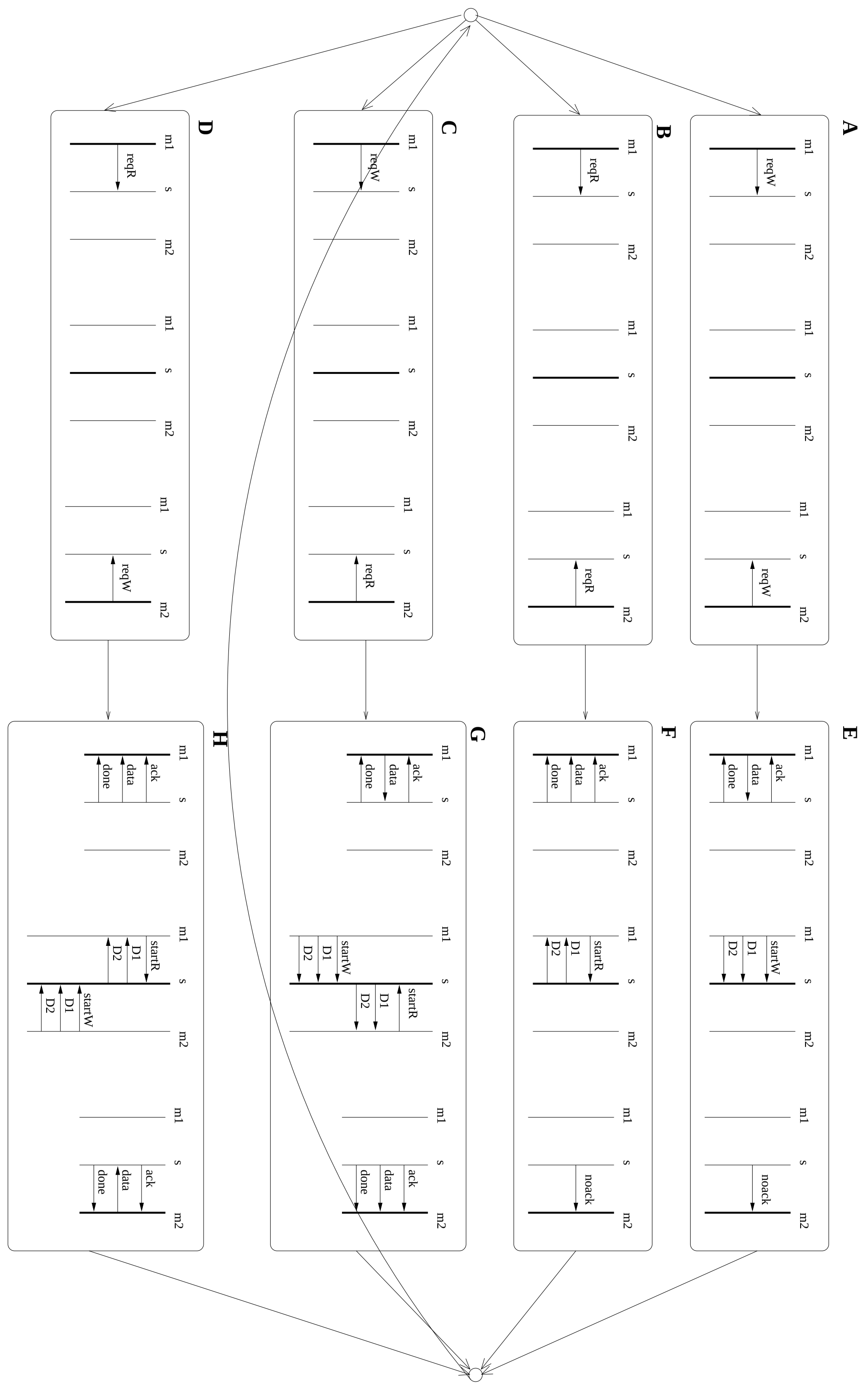


Figure 5.5: Two masters and one slave with read/write overlap

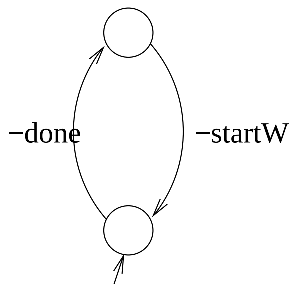


Figure 5.6: Behavioral requirement for overlap read/write example

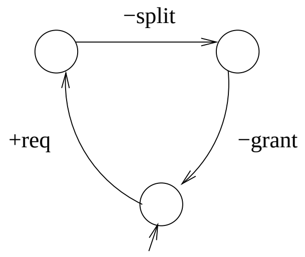


Figure 5.7: Behavioral requirement for split transaction example

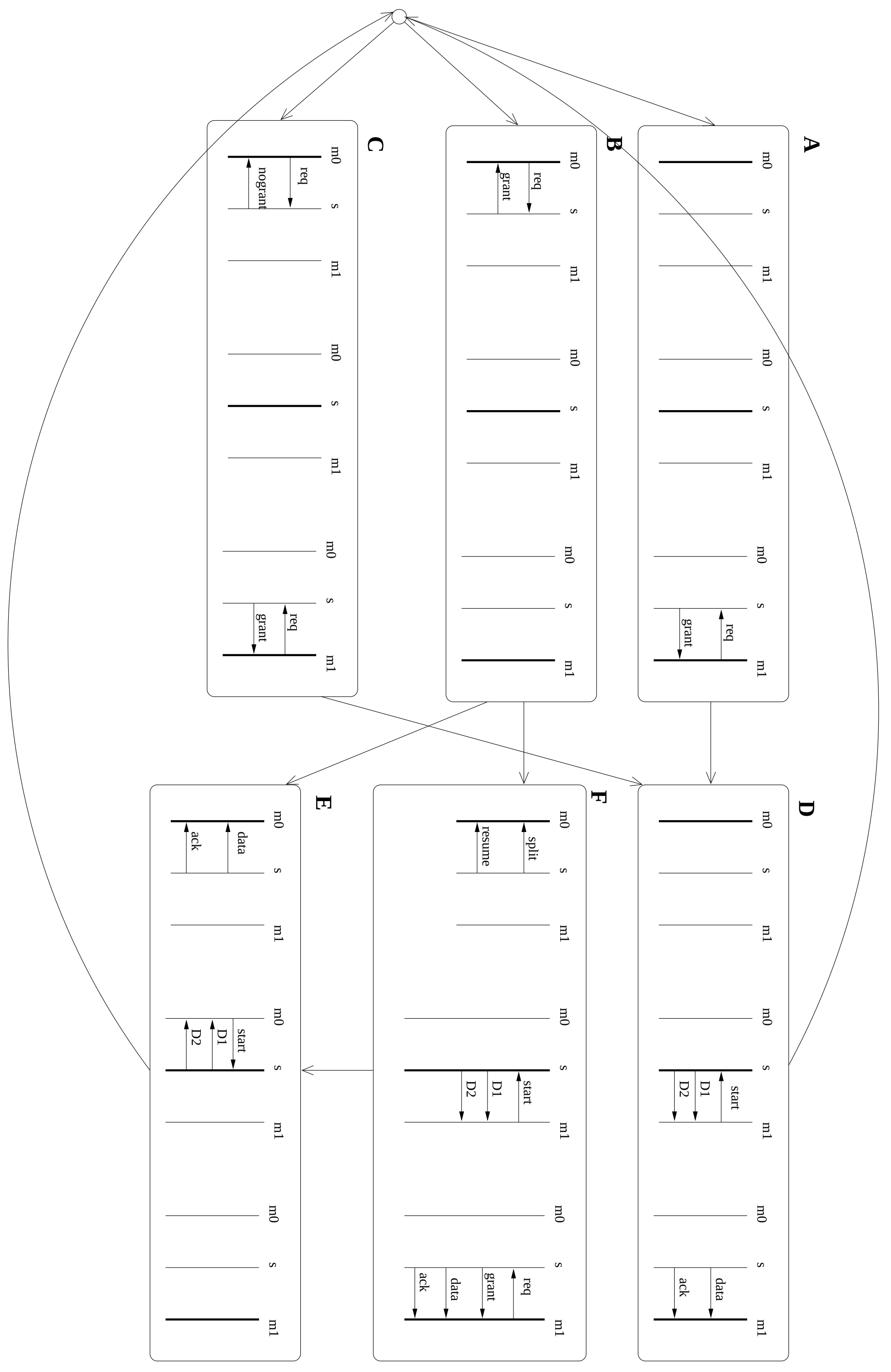


Figure 5.8: Two masters and one slave with split transaction

Chapter 6

Implementation

6.1 Overall System Design

In this chapter, we describe the SystemC-based implementation of our converter generator.

SystemC – viewed as a programming language – is a collection of class libraries built on top of C++ and hence is naturally compatible with the object-oriented design philosophy. It allows both applications and platforms to be expressed at high levels of abstraction, while enabling the linkage to hardware implementation and performance evaluation. The semantics of SystemC has been standardized through a kernel simulator. For more background information concerning SystemC, we refer the reader to (Arnout, 2000; Grotker et al., 2002; SystemC).

Our tool takes as input an HMSC description of interactions among multiple components. This description comes in two parts: the HMSC description and the converter specifications. The HMSC description includes the graph structure of the HMSC and the component views (or MSCs) inside each node. The converter specifications are comprised of message relationship specifications and behavioral specifications. Our technique automatically generates a converter in the form of SystemC code. In our scheme, the component views are also specified in SystemC, and hence we can compile the converter along with the component views. The resultant system can be simulated using the SystemC simulation kernel. The simulation is driven by a path through the HMSC, which we provide as an input for the simulation. The converter generator is written in C++ and its structure is shown in Figure 6.1. We, next, look at the format of input files in details.

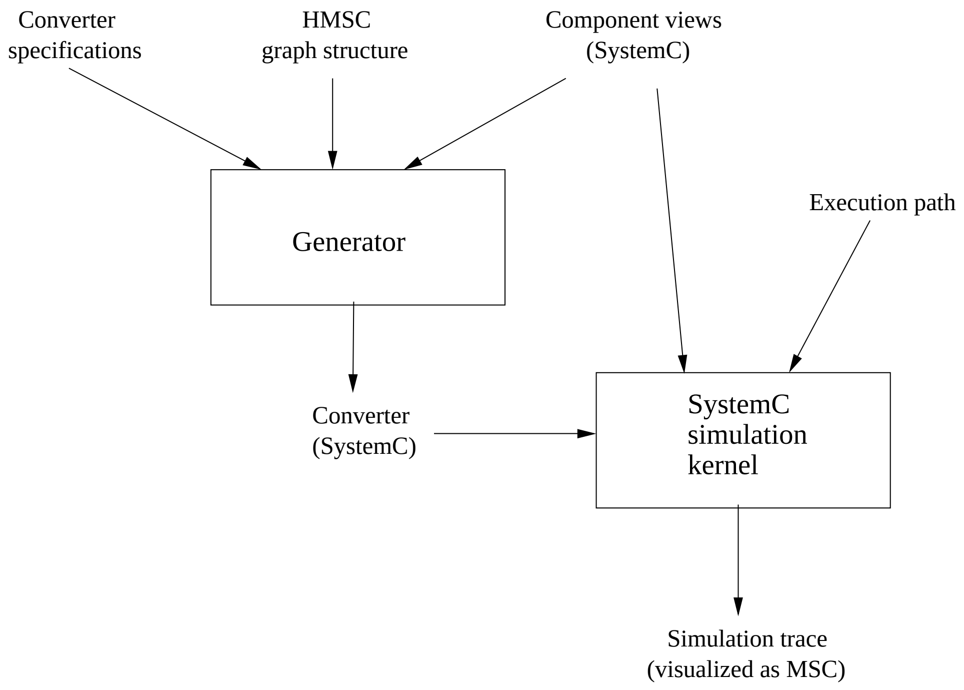

Converter specifications
HMSC graph structure
Component views (SystemC)
Generator
Execution path
Converter (SystemC)
SystemC simulation kernel
Simulation trace (visualized as MSC)

Figure 6.1: Overall structure of the implementation

6.2 Input Formats

The first part of the input, the converter specifications are described in a text file comprised of all message relationship specifications and behavioral specifications. The converter specification description file of the example in Figure 5.5 is shown below.

```
DATA=(D1OneE,dataOneE):(D2OneE,dataOneE)
DATA=(dataOneF,D1OneF):(dataOneF,D2OneF)
DATA=(D1OneG,dataOneG):(D2OneG,dataOneG)
DATA=(dataTwoG,D1TwoG):(dataTwoG,D2TwoG)
DATA=(D1TwoH,dataTwoH):(D2TwoH,dataTwoH)
DATA=(dataOneH,D1OneH):(dataOneH,D2OneH)
CONSTRAINT=state1:state1 doneOneG state2:state2 startReadTwoG state1:
CONSTRAINT=state1:state1 doneTwoH state2:state2 startReadOneH state1:
```

Each line describes a specification; a message relationship specification has a corresponding line marked with DATA, and a behavioral specification has a corresponding line marked with CONSTRAINT. For instance, the first line of the above file says that in the scenario E *data* message in the $master_1$ view will be chopped into D_1 and D_2 messages in the *slave* view. The last line of the above file describes a behavioral automaton/specification that requires in the scenario H, the *done* message must be sent to $master_1$ before $start_{Read}$ signal is sent to *slave*. This last line is, basically, the textual description of the automaton in Figure 5.6.

The second part of the input, the graph structure of an HMSC (*i.e.* the name of nodes and the edges between nodes), is described in a simple text file; and the views of the components (*i.e.* the MSCs inside each node), is specified in SystemC in the following way. All components described in the HMSC are presented as SystemC modules that exchange messages through FIFO channels. For each module representing a component, a number of ports are specified, one for each message in MSCs of that component. The port declarations only differ in types of data transmitted through them. For a control message, the type of the corresponding port is boolean; while for a data message, it can be a SystemC data type or a user-defined one. The view of any component p at any node n of the HMSC is encoded as a SystemC function in the module representing p. The body of this function is just a list of write and read port actions corresponding to send or receive message actions along the life-line of p in the corresponding MSC. In addition, in each module there is a main thread which interacts with the environment thread. It will call the corresponding function when the control point of the component reaches a node in the HMSC. A snippet of the SystemC module representing the master component of the example in Figure 5.1 is shown below.

```
// SystemC Module
SC_MODULE(master) {
  // Port declaration
  sc_fifo_out<bool> reqA;
  sc_fifo_in<bool> nograntB;
  sc_fifo_out<sc_int<32>>transferB;
  sc_fifo_in<bool> grant;
  ...
  // Constructor
  SC_CTOR(master){
        SC_THREAD(main_action);
  }
 // main thread
 void main_action(){
   char* next = path->nextNode();
   while(strcmp(next,"")!=0){
      if (strcmp(next,"A") == 0) NodeA();
      if (strcmp(next,"B") == 0) NodeB();
      if (strcmp(next,"C") == 0) NodeC();
      if (strcmp(next,"D") == 0) NodeD();
      ...
      next = path->nextNode(); }
 }
 // protocol of master at node B
 void NodeB() {
      bool b = nograntB->read();
      transferB->write(data);
 }
 // protocol of master at node C
 void NodeC() {
      bool b = grant->read();
      //request signal sent
      transferC->write(data);
      //data sent
 }
```

6.3 Generated Protocol Converters

The generated converter is a SystemC module too. The converter is multi-threaded. As discussed in Section 6.1, the converter threads interact with the environment thread (a simulation path in our implementation). We show here only the task of the converter thread dealing with master component in node C of the example in Figure 5.1.

```
void node_C_master(){
  while(true){
    wait(event_C_master
       | termComp[0].value_changed_event());
    if(termComp[0].read()) return;
    wait(clock->posedge_event());
    // send grant message
    port_master_grantC->write(true);
    time = clock->time_stamp().value();

    if (port_master_transferC->num_available()==0)
    wait(port_master_transferC.data_written_event());
    // receive transfer message
    sc_int<8> d = port_master_transferC->read();
    q_transferC->push(d);
    time = clock->time_stamp().value();

    chooseNextNode(...);
   }
}
```

Basically, a converter thread dealing with a component in a node is a nonterminating `while` loop - this is the standard way in SystemC to model threads; at the beginning of the loop the converter thread checks to see (1) if it is allowed to execute and (2) if the simulation is finished. If the simulation is finished, the thread just terminates; otherwise, it will execute the body of the loop. As we described in the Section 4.2 the actions in the body of the loop are just the dual-view of the component view which it interfaces. At the end of the loop, the thread will check with the *RUN* channel to get which converter thread of a node it should pass control to.

A behavioral specification is implemented as a monitor. The converter will check with the monitor before executing any action which sends or receives a message which is in a alphabet of a behavioral specification, and it will update the monitor after executing the action. The following snippet of codes shows the implementation details:

```
while ( !isMoveable("splitZeroF") )
{
  wait( clock->posedge_event() );
}
port_masterzero_splitZeroF->write(true);
doAct("splitZeroF");
```

Also based on message relationship specifications, the converter will automatically handle dynamic data-sizing. The following is the snippet of codes merging message D_1 and D_1 from the *slave* to send them to $master_0$ as *data* message in the scenario E of the example shown at Figure 5.5.

```
// wait for D1 and D2
while (q_dataZeroE_D1SlaveE->empty()) {
   wait(cycle, time_unit);
}
while (q_dataZeroE_D2SlaveE->empty()) {
   wait(cycle, time_unit);
}
sc_int<32> d_D1SlaveE = q_dataZeroE_D1SlaveE->front();
q_dataZeroE_D1SlaveE->pop();
sc_int<32> d_D2SlaveE = q_dataZeroE_D2SlaveE->front();
q_dataZeroE_D2SlaveE->pop();
sc_int<32> d_masterzero_dataZeroE;
// merge D1 and D2 to have data
d_masterzero_dataZeroE = (d_D1SlaveE,d_D2SlaveE);
// send data message to the master zero
port_masterzero_dataZeroE->write(d_masterzero_dataZeroE);
```

Timing constraints and clock sensitivities Since SystemC supports clock sensitivities and timers, we have augmented our HMSC specification with these features. This ensures that the results obtained by simulating the synthesized converter over the SystemC simulation kernel will be more accurate (in terms of number of clock cycles to execute a sequence of transactions). In particular, we allow any send/receive/internal event inside any node of the HMSC to be guarded by two special conditions: *pos_edge* and *neg_edge*. *pos_edge* is true at each positive edge of the system clock and *neg_edge* is true at each negative edge of the system clock. As a trivial example, if two events e_1, e_2 in an MSC are both guarded by *pos_edge* and e_1 happens-before e_2 (as per the partial order of the MSC), then there must be at least one clock cycle delay between e_1 and e_2.

In our implementation, we also allow for one timer per process in the HMSC specification. The timer may be set, reset, counted down or timed out. This allows us to specify a bound on the delay between two events within a process/component. We note that several other approaches are possible for specifying timing constraints in the MSC-based specifications (*e.g.* see (Harel and Marelly, 2002)). Incorporating these mechanisms in our converter generation toolkit is a topic of future work.

6.4 Implementation Framework

We present the framework of our implementation in this section. The following is the structure of the working directory:

- **data**\ this folder contains the code-templates to generate the protocol converter and the code-templates to produce the simulation file.
- **input**\ this folder contains all the input files which are the converter specifications, the HMSC graph structures, and the component views in form of SystemC code.
- **output**\ this folder contains the generated protocol converters, and all other supplementary modules to run the simulation.
- **generator**\ this folder contains the generator program.

- **parser\** this folder contains the parser to analyze the component views in SystemC codes. The parser is written using BISON and LEX.
- **util\** all other support modules.

The workflow The workflow of the system is as follows. To generate the protocol converter the generator `generator\gen` reads the file `input\Generator.ini` to get all parameters. The `ini` file provides all the information for the generator to produce the protocol converter which will be put at `output\Converter.h`. The generator will also generates the `output\Main.cpp` which pulls all the necessary modules together to run the simulation. To run the simulation, using the `output\Makefile` to compile the simulation. A executable file `converter.x` will be produced; running it will produce the simulation trace as `output\msc.fig`.

The structure of the `ini` file contains multiple pair of `property=value`.

- `modulesFile=` the file contains component views in SystemC.
- `graphFile=` the file contains the structure of the HMSC.
- `SimPath=` the file specifies the simulation path which is a list of HMSC node following the structure of the HMSC graph.
- `ConstraintFile=` the file specifies converter constraints.
- `converterFile=` the file in which the generated protocol converter will be stored.
- `mainFile=` the file in which the simulation module will be stored.
- `converterSkel=` the template used in the converter generation process.
- `mainSkel=` the template used to generate the simulation file.

- `time_output=` if the value is `true`, the converter will time stamp the log file.
- `clock_sensitivity=` if set to `true`, the actions will be guarded by clock edges.
- `time_unit=` the SystemC time unit to be used. Normally set to `SC_PS`.
- `cycle=` the smallest time unit to be used.

Generation Algorithm We now present how we implement the generator. The generator contains two big tasks.

```
int main() {
// Create the parse tree of the component views in SystemC
CreateComponents();

// Generate the protocol converter
GenerateConverter();
}
```

Parsing the SystemC code is straightforward because we restrict the format of the code as explained in Section 6.2. The general converter structure is fixed and what differs among the converters is the converter threads communicating with the components. Therefore, we actually pre-build a code-skeleton for the protocol converter and insert only the missing parts when generate a new protocol converter.

The general structure of a protocol converter is presented below.

```
SC_MODULE(Converter)
{
  // internal queue declaration - generated
  queue<sc_int<32> >* q_dataZeroE_D1SlaveE;
  queue<sc_int<32> >* q_dataZeroE_D2SlaveE;
  ....

  // port declarations - generated
  sc_fifo_in<bool > port_masterzero_reqZeroB;
  sc_fifo_out<bool > port_masterzero_grantZeroB;
  sc_fifo_in<bool > port_masterzero_reqZeroC;
  ....

  // thread enable events - generated
  sc_event event_A_masterzero;
  sc_event event_D_masterzero;
  sc_event event_B_masterzero;

  SC_CTOR(Converter)
  {
    // Initialize internal queues
    // Initialize thread enable events
    // Initialize all converter threads
    // Take in the simulation path
  }

  // Converter threads for each node
  void node_A_masterzero(){
    .....
  }

  void node_D_masterzero() {
    ....
  }

}
```

The most important aspect of the generator is to generate the converter threads in each HMSC node. In each HMSC node, we have a separate converter node talking to each component. The algorithm to generate those threads is presented below.

```
void generateConverterThread(component, HMSCNode) {
 // Get the life-line of the component in the HMSCNode
 GetMSCLine(component, HMSCNode);

 foreach message in the MSCLine {
   if (message->direction() == incoming) {
     // generate converter outing message
     if (message->isNormalMessage()) {
       // this message is not specified in the converter specification
```

```
        // Generate the converter code such that
        // it just send out the same message to the component
      }
      else if (message->isConstraintMessage()) {
        // This message is constrained by the extra automata
        // but it is not a data message
        // Generate the converter code such that
        // the converter will consult the automata before
        // send out the message to the component
      }
      else if (message->isTranferMessage())  {
        // This message is data message which need transfered among
        // components
        // Generate the converter code such that
        // the converter check the queue to see if
        // the corresponding message has come; if yes, send out the
        // message to the component, otherwise wait.
      }
    } else { // outgoing
       // generate converter incoming message
       // The algorithm is similar to the above incoming message case.
    }
  }
}
```

The generator is based heavily on the template, hence the algorithm is quite strait forward, most of the details are pre-built in the template codes. We have so far through the code snipets shown all the aspects of the generation process. For more details and further information, readers can refer to the actual implementation.

Chapter 7

Discussion

In this chapter, we summarize our contributions of the work in this thesis and suggest some future research directions.

7.1 Summary of the Thesis

In this thesis, we have investigated the problem of automatically generating converters which enable communication among embedded system components using incompatible protocols. An important feature of our work is that it is based on scenario-based descriptions of component interactions. Given the overall component interaction patterns of the system in form of an HMSC, we automatically generate a multi-threaded protocol converter in SystemC. This allows us to exploit the SystemC simulation kernel for simulating the converter along with component interfaces at a fairly high level of abstraction. We summarize our main contributions below.

- To the best of our knowledge, our work is the first one to study the problem of synthesizing protocol converters using scenario-based descriptions. Our approach has many advantages over previous works. For instance, HMSC-based descriptions can naturally and intuitively present the interaction among multiple parties. We can model complex systems comprising of components involved in multiple *mode of interactions* in which control/data signals can flow in both directions. Based on the SystemC simulation engine, we can easily simulate the generated protocol converter along with the component interfaces. For a more in-depth comparison, please refer to Section 3.2.

- We have built a complete system to conduct experiments to test the effectiveness of our approach. Our generator is written in C++, and it produces converters in SystemC which can be compiled with the component views for simulation. The system also provides MSC-descriptions of result simulation traces for easy inspection.

- We have extended our previous work to handle dynamic data sizing in which the converter need to combine/chop data packets. In order to have this capability we introduce the use of message relationship specifications.
- We incorporate behavioral specifications in form of automata as the mean for imposing additional behavioral constraints on the generated protocol converters. This allows us to model more complicated systems.

7.2 Future Work and Concluding Remarks

In terms of future work, there exist various opportunities for extending our converter generator's capabilities.

- One direction will be to specialize the converter generation technique to handle software/hardware interfaces. In a larger context, the work initiated here has a bearing on hardware/software co-design. If each component is chosen to be realized fully in hardware or software, our scenario-based description clearly identifies the communication interfaces. Consequently, our approach, especially after the enhancements suggested above, will rapidly yield an executable description of the interconnect fabric.
- Another line of work is to study and extend the scenario-based descriptions themselves. Instead of using standard HMSCs, we can use similar formalisms such as Communicating Transaction Processes (CTP) (CTP; Roychoudhury and Thiagarajan, 2002, 2003). Since an executable specification can be extracted from CTP in a straightforward manner, it can potentially make it

easier to generate an executable description of the interconnect fabric. One limitation of our our current work is that a converter needs a environmental thread to guide the simulation. The idea is that by exploring some other forms of scenario-based descriptions we can eliminate the need for an environmental thread.

Bibliography

J. Akella and K. McMillan. Synthesizing converters between finite state protocols. In *International Conference on Computer Design*, 1991.

CoreConnect Bus Architecture. `http://www-306.ibm.com/chips/products/coreconnect/`.

AMBA. *AMBA On-chip Bus Specification*. ARM Limited, 1999.

Guido Arnout. SystemC standard. In *Asia South Pacific Design Automation Conference*, pages 573–578. IEEE Computer Society Press, 2000.

R.A. Bergamaschi and William R. Lee. Designing systems-on-chip using cores. In *Design Automation Conference*, pages 420–425. ACM Press, 2000. URL `citeseer.ist.psu.edu/article/passerone97automatic.html`.

Gaetano Borriello. *A New Interface Specification Methodology and its Applications to Transducer Synthesis*. PhD thesis, University of California, Berkeley, 1988.

Gaetano Borriello, Luciano Lavagno, and Ross B. Ortega. Interface synthesis: a vertical slice from digital logic to software components. In *International Conference on Computer Aided Design*, pages 693–695. IEEE Computer Society Press, 1998.

Chinook. The Chinook project. World Wide Web, `http://www.cs.washington.edu/research.projects/lis/www/chinook/`, 1999.

CTP. System level design methods for reactive embedded systems. World Wide Web, `http://www.comp.nus.edu.sg/~ctp/`.

Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 109–120. ACM Press, 2001. URL `citeseer.nj.nec.com/dealfaro01interface.html`.

Peter Flake and Simon J. Davidmann. Superlog, a unified design language for system-on-chip. In *Asia South Pacific Design Automation Conference*, pages 583–586. IEEE Computer Society Press, 2000.

D.D. Gajski, J. Zhu, and R. Domer. *Specification Language: SpecC: Design Methodology*. Kluwer Academic Publishers, 1997.

T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *IEEE/ACM Intl. Symp. on Modeling, Analysis and Simulation of Comp uter and Telecommunication Systems (MASCOTS)*, 2002.

D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Pl ay-Engine*. Springer-Verlag, 2003.

D. Harel and P.S. Thiagarajan. Message sequence charts. In L. Lavgno, G. Martin, and B. Selic, editors, *UML for Real: Design of Embedded Real-time Systems*. Kluwer Academic Publishers, 2003.

Metropolis Project. Metropolis: Design environment for heterogeneous systems, 2001. Details available from `http://www.gigascale.org/metropolis/`.

S. Narayan and D.D. Gajski. Interfacing incompatible protocols using interface process generation. In *Design Automation Conference (DAC)*, 1995.

Roberto Passerone, Luca de Alfaro, Thomas A. Henzinger, and Alberto L. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *International Conference on Computer Aided Design (ICCAD)*. IEEE Computer Society Press, November 2002. URL `citeseer.nj.nec.com/article/passerone02convertibility.html`.

Roberto Passerone, James A. Rowson, and Alberto L. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Design Automation Conference*, pages 8–13. ACM Press, 1998. URL `citeseer.ist.psu.edu/article/passerone97automatic.html`.

Polis. A framework for hardware-software co-design of embedded systems. World Wide Web, `http://www-cad.eecs.berkeley.edu/Respep/Research/hsc.abstract.html`, 1997.

A. Roychoudhury and P.S. Thiagarajan. An executable specification language based on message sequence charts. *Formal Methods at the Crossroads: From Panacea to Foundational Support, Springer Lecture Notes in Computer Science*, 2002.

A. Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. In

IEEE International Conference on Application of Concurrency in System Design (ACSD). IEEE Computer Society Press, 2003.

A. Roychoudhury, P.S. Thiagarajan, Tuan-Anh Tran, and Vera A. Zvereva. Automatic generation of protocol converters from scenario-based specifications. In *Proc. 25th IEEE International Real-time Systems Symposium (RTSS 2004)*. IEEE Computer Society Press, 2004.

Alberto Sangiovanni-Vincentelli, Marco Sgroi, and Luciano Lavagno. Formal models for communication-based design. In *Proceedings for International Conference on Concurrency Theory (CONCUR '00)*. Springer-Verlag, August 2000.

Alberto L. Sangiovanni-Vincentelli, Patrick C. McGeer, and Alexander Saldanha. Verification of electronic systems. In *Design Automation Conference*, pages 106–111. ACM Press, 1996. URL `citeseer.ist.psu.edu/141312.html`.

Alberto L. Sangiovanni-Vincentelli and James A. Rowson. Interface-based design. In *Design Automation Conference*, pages 178–183. ACM Press, 1997. URL `http://portal.acm.org/citation.cfm?id=266060&dl=ACM&coll=portal`.

SystemC. SystemC community. World Wide Web, `http://www.systemc.org/`, 2003.

SystemVerilog. SystemVerilog. World Wide Web, `http://www.systemverilog.org/`, 2005.

Tuan-Anh Tran. Qualifying examination term paper. Technical report, National University of Singapore, 2004. Available for downloading from `http://www.comp.nus.edu.sg/~trantuan/termpaper.ps`.

S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *9th European Software Engineering Conferece and*

9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE), 2001.

Rosetta Website. Rosetta: Language support for systems level design. World Wide Web, `http://www.ittc.ku.edu/Projects/rosetta/index.html`, 2005.

Z.120. Message Sequence Charts (MSC'96), 1996.

V.A. Zvereva. Converter between incompatible protocols specified using MSC diagrams. Technical report, National University of Singapore, 2004. Available from `http://www.comp.nus.edu.sg/~abhik/pdf/Converter.pdf`.

Name:	Tran Tuan Anh
Degree:	Master of Science
Department:	School of Computing
Thesis Title:	Automatic Generation of Protocol Converters from Scenario-based Specifications

Abstract

With the complexity of the current embedded systems, it is impossible for a single company to design and manufacture and entire electronic system in time an within reasonable cost. Hence design re-use based on pre-designed intellectual property (IP) cores has become an absolute necessity for embedded system companies to remain competitive. However, one of the difficulties in IP reuse is the incompatibilities between the protocols used by various parts. Hence, reusing IP cores often requires designing converters (glue-logic) to enable their communication. In this work, we study the problem of automatically generating a protocol converter which enables various embedded system components - using different (possibly incompatible) protocols - to talk to each other. The novelty of our approach is that it takes as input a scenario-based description of inter-component interactions described as a collection of Message Sequence Charts. From this specification, we systematically synthesize, when possible, the protocol converter that lets the components to use their native protocols while overall pattern of interaction is correctly realized. We demonstrate the feasibility of our approach by modeling some important features of existing Systems-on-Chip bus protocols.

Keywords:
Embedded Systems, Protocol Converters, Scenario-based Descriptions, Incompatible Interfaces, Hardware-Software Codesign, System Level Design

AUTOMATIC GENERATION OF PROTOCOL CONVERTERS FROM SCENARIO-BASED SPECIFICATIONS

TRAN TUAN ANH

NATIONAL UNIVERSITY OF SINGAPORE

2005

Automatic Generation of Protocol Converters from Scenario-based Specifications Tran Tuan Anh 2005