Automatic Generation of Protocol Converters from Scenario-based Specifications

Abhik Roychoudhury P. S. Thiagarajan Tuan-Anh Tran
School of Computing, National Univ. of Singapore
{abhik,thiagu,trantuan}@comp.nus.edu.sg

Vera A. Zvereva*
State Univ. of St. Petersburg, Russia
vera@vz8760.spb.edu

Abstract

Reuse of IP blocks is an important design philosophy for embedded systems. This allows shorter design cycles under tight time-to-market constraints. However, reusing IP blocks often requires designing converters (glue logic) to enable their communication. In this paper, we study the problem of automatically generating a protocol converter which enables various embedded system components (possibly with incompatible protocols) to talk to each other. Our work takes as input, a rich description of inter-component interactions described as a collection of Message Sequence Charts. We then automatically synthesize from this input a protocol converter in SystemC. 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 modelling some simplified bus protocols that capture key features of existing System-on-Chip bus protocols. We then generate the bus controller as the protocol converter.

1. Introduction

Due to the very large gate counts available on a single chip, SoC (*S*ystem *o*n a *C*hip) designs have become feasible. The accompanying economic pressures make it imperative that one does not start from scratch each time to design such systems. Instead, one needs a methodology by which predesigned components -often called IP cores in this context- can be quickly integrated to produce correct implementations. 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, one IP core may have been designed on the basis of a "push" protocol for transferring data in which it emits a *ready* signal followed by the transmission of a piece of data whereas the component with which it is trying to communicate may have been designed on the basis of "pull" protocol where it sends out a *free* signal followed by the reception of a piece of data. One must design glue logic using which such pairs of incompatible components can be connected together. The role of this logic is to let the components to use their native protocols and be oblivious to the incompatibility of the protocol being used by its communication partner(s).

In this paper, we investigate the use of *scenarios* for the specification and realization of such glue logics. We show how the popular visual formalism of Message Sequence Charts (MSCs) and High-level Message Sequence Charts (HMSCs) [20] 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 glue logic called a *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.

Intra- and Inter-component view in System Level Design A key requirement of a component-based design methodology is the clear separation between computational and communication features of an application [9, 10, 14]. 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.

∗ This work was done during a research internship at the School of Computing, National University of Singapore.

The relationship between intra-component and inter-component (scenario-based) system descriptions is complex and subtle [12]. 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 [13] and the references therein). In this paper, we assume that we have been given an HMSC-based description of the component interaction patterns and that it is consistent with the intra-component specification; since the intra-component specification is not relevant for our purposes it is not displayed.

MSCs/HMSCs for describing Inter-component view In the inter-component description of a system, we view an MSC as an atomic 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. For example, consider a node called *transfer* denoting transfer between a master m and slave s. This node will typically contain two MSCs Ch_m and Ch_s; Ch_m describes the *master*'s view of *transfer* and Ch_s captures the *slave* component's view of *transfer*.

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 signals in case the signal alphabets of the interacting components are not identical (as is often the case even after taking into account mapping of signal names across components). Secondly, the converter will store data in order to resolve incompatibilities. Last but not the least, we allow the converter to speculatively generate control signals in advance to resolve potential deadlocks. This technique is based on the implicit hypothesis that in the present setting, there is no hidden data or control flow between the components other than what is captured by the HMSC specification. In more general settings, which we do not consider here, the converter will not generate speculatively control signals, for example, when these signals are a part of a protocol ensuring mutual exclusion. As might be expected, the converter will also not generate data values speculatively. Hence our synthesis procedure will fail if there is two-way transfer of data which is incompatible. We provide a syntactic check for detecting such incompatibilities. In the present version of the converter we do not deal with data format incompatibilities such as 16 bit sends vs. 8 bit receives etc. However our method can be easily extended to handle such incompatibilities.

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 is 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 as an MSC. Due to translation into SystemC we can introduce clock sensitivities and timers in our input specification language.

Section Organization In the next section, we survey related work on the converter generation problem. In Section 3 we briefly recall the basic features of MSCs and HMSCs. In Section 4, we formulate the converter generation problem and describe, using illustrative examples, our solution. In the next two sections we describe our implementation and its application to a detailed example inspired by SoC bus protocols such as AMBA [2]. Section 7 concludes the paper with a discussion on directions for future research.

2. Related Work

The problem of generating glue logic for protocol conversion has been studied in the past. Borriello [3] generated glue logic between two circuit blocks (whose interface behavior is captured by timing diagrams) via transducer synthesis. The work of [1] describes protocols as finite state machines and develops the protocol converter from the product machine. Narayan and Gajski [15] 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, we synthesize 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 a scenario-based description. Previous works that have influenced our research have been carried out in intra-component settings [17, 6, 16]. 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. As a result, these automata are often referred to as *interface* automata (see [7, 4, 8]). In [17], the focus 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. There is a single data token that is transferred and both the components, specified as regular expressions, are assumed to be driven synchronously by a common clock. In [6], the focus is on interface automata modeling interactions between software components. Each interface automaton has input, internal and output actions and it is possible to syntactically distinguish between environment supplying inputs and a component supplying inputs. The main focus is on determining whether two components can interact in a compatible way - in the presence of the constraints imposed by the individual interface automata- with the proviso that the external environment may be constrained to ensure compatibility. No attempt is made to insert a converter to resolve incompatibilities. This is done in [16] which is in some sense a common generalization of [17] and [6]. A nice feature here is additional behavioral constraints may be imposed on the converter using a specification automaton. The focus is however mainly on one-way flow of information between two components with matching alphabets. The two components are deemed to be not *adaptable* (a converter does not exist) if it is impossible to avoid reaching a joint state of the two components in which the converter can not generate a response that is compatible with the protocols of the two components. Again, the two components are assumed to work synchronously. Thus the existing works on 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 works.

3. MSCs and HMSCs

We recall the basic notions/notations concerning MSCs and HMSCs. A more detailed survey appears in [13].

An MSC describes a snippet of behavior involving multiple components interacting with each other. In its simplest form (which is the one we shall work with in this paper), the components communicate with each other through FIFOs. The visual representation of an MSC is shown in Figure 1. The vertical lines, often referred to as lifelines (instances) capture the behavior of the components. The horizontal edges capture a communication; the origin of the edge is the sender, the target of the arrow is the receiver and the label associated with the edge is the message that is being communicated. The darkened rectangular boxes are the events associated with the MSC. In what follows we will often refer to an MSC as a chart. The example shown in Figure 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 tracking the number of times the user has gained access to R.

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. 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). 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***.

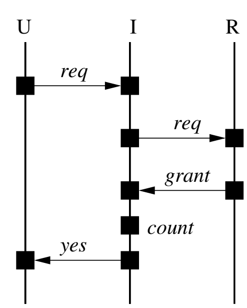


Figure 1: A simple MSC

One standard mechanism [20] 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, the specification can be hierarchical, in that a state of the automaton can be la-

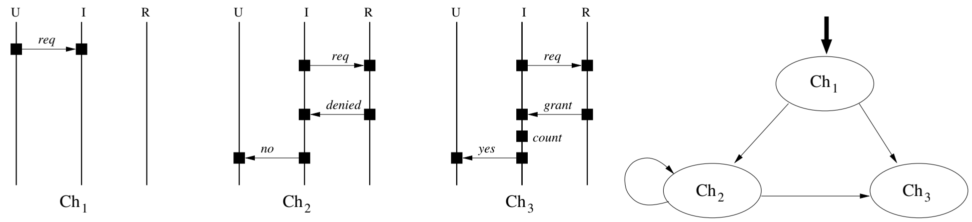


Figure 2: A simple HMSC

beled by an HMSC instead of an MSC. In this paper, we shall ignore this feature and instead consider flattened HMSCs. An example of an HMSC is shown in Figure 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 keeps trying until it gets the response ”granted”, at which 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 the present paper, 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, $Ch_1.Ch_2$ 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 executing the 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. The second way of concatenating charts - which is the one we will consider in this paper - is the asynchronous concatenation. Here the concatenation is carried out at the level of life-lines. In Figure 3 we show the chart Ch_{23} obtained via the asynchronous concatenation $Ch_2 \cdot Ch_3$ where Ch_2 and Ch_3 are as shown in Figure 2. Note that the receipt of `no` in Ch_2 can take place after the sending of `req` in Ch_3 under asynchronous concatenation.

The meaning of a 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; recall that 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, the chart $Ch_1 \cdot Ch_2 \cdot Ch_2 \cdot Ch_3$ is in the collection while $Ch_1 \cdot Ch_3 \cdot Ch_2$ is not.

We conclude by introducing the notion of **cycle bounded executions** of an HMSC. This notion will be used to restrict

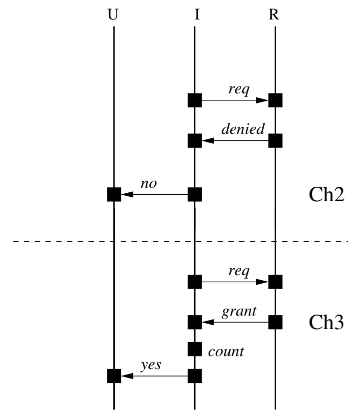


Figure 3: Concatenation of MSCs

the amount of overtaking allowed between processes in our SystemC implementation. Suppose

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

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 $\pi = s_i s_{i+1} \dots s_{n+i}$ constitutes a cycle in the HMSC. Let Ch_j be the chart associated with s_j for each j in $\{0, 1, \dots n+i\}$ and $Ch = Ch_0 \cdot Ch_1 \cdot \dots Ch_m$. 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.

4. Technique

In this section, we outline our technique for automatically generating protocol converters. We first present the 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. The protocols executed by the network of components is assumed to be described as an HMSC but where each node will have a *set* of MSCs associated with it; one MSC for each process taking part in the mode of interaction associ-

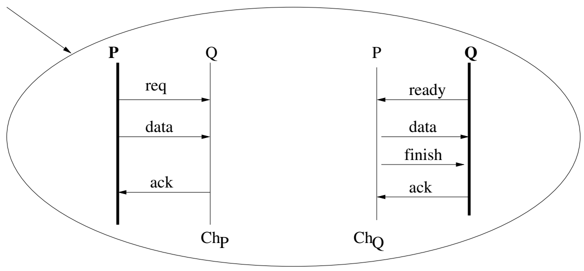


Figure 4: A simple HMSC specification of incompatible protocols

ated with that state. *For convenience, we will refer to this extended notion too as an HMSC.*

We show a simple example of an HMSC in Figure 4. 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.

$$\langle P!Q, req\rangle, \langle P!Q, data\rangle, \langle P?Q, ack\rangle \; (1)$$
$$\langle Q!P, ready\rangle, \langle Q?P, data\rangle, \langle Q?P, finish\rangle, \langle Q!P, ack\rangle \; (2)$$

In the rest of the paper, we assume that each node n of the 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) above is the P-view and (2) is the Q-view of the HMSC node shown in Figure 4. 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 also important, particularly for detecting inherently incompatible protocols. We discuss this issue later in this section.

4.1. Converting 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. Our presentation here is an informal one; a formal description appears in [21]. Let the agents of the single node be $p_1, \ldots, p_k$ with Ch_{p_i} being denoted for convenience as Ch_i.

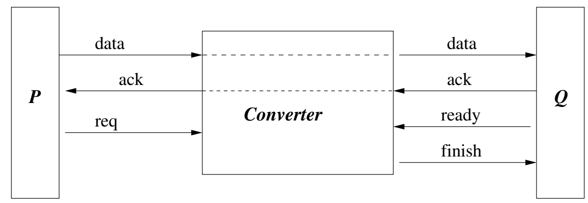


Figure 5: Inputs/Outputs of Protocol Converter for Figure 4

First, we partition the whole space of signals into *control signals* and *data signals*. Secondly, both control and data signals can be further classified as follows.

- **Shared Signals**: Signals which are sent by one process p_i according to Ch_i and received by the corresponding process p_j according to Ch_j. *e.g.* the signals `data` and `ack` in Figure 4.
- **Input Signals**: Signals which are sent by one process p_i according to Ch_i but not received by the corresponding process p_j according to Ch_j. *e.g.* the signals `req` and `ready` in Figure 4.
- **Output Signals**: Signals which are received by one process p_i according to Ch_i but not sent by the corresponding process p_j according to Ch_j. *e.g.* signal `finish` in Figure 4.

Based on the above categorization, we can identify the inputs and outputs of the desired protocol converter for the specification of Figure 4. To distinguish the shared signals from the input/output signals, we show the shared signals via dotted lines inside the box corresponding to the converter (refer Figure 5). The *Input* signals will be consumed by the and *Output* signals will be autonomously produced by the protocol converter. As for the shared signals, there are several possibilities. One strategy is to wait to receive any such signal m from the sender process p_i and only then forward m to the receiver process p_j. A more aggressive policy will be to send the signal m to p_j even before it is received from p_i. Thus we can allow speculative sends of shared control control signals. However we will not be able to speculatively send shared data signals since these will, in general have values associated with them that can not be guessed. In the example of Figure 4, `data` is a shared data signal and `ack` is a shared control signal.

We note that our definition of shared signal could lead to name clashes in signal names, since more than two components are involved in a HMSC node in general. For convenience, we assume that signal name clashes do not occur (*i.e.* clashes are avoided through renaming if necessary).

We now show the protocol converter generation for the one node HMSC specification in Figure 4. For simplicity, we generate a converter which does not send any shared signal before receiving it.

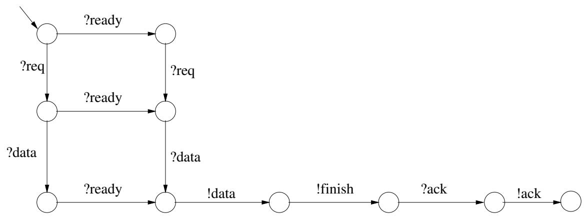
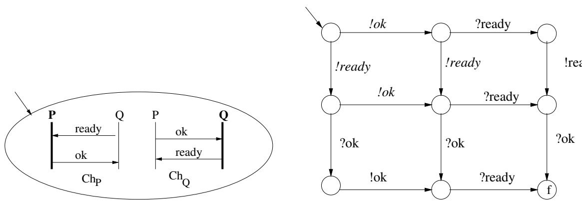


Figure 6: Protocol converter for specification of Figure 4



(a) (b)

Figure 7: (a) A protocol specification which seems incompatible (b) A converter for the specification in (a)

- It first receives the input signals `req` and `ready` which may arrive in any order.
- After `req` is received, the converter waits to receive the shared signal `data` from P; after `data` is received, it is sent to Q.
- At this stage, both P and Q are expecting to receive signals `finish` and `ack` respectively. The converter generates the output signal `finish` and send its to Q.
- Finally, when the converter receives `ack` from Q, it sends it to P.

The transition system corresponding to the protocol converter for Figure 4 appears in Figure 6. 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 the 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 $s_{1,n_1} \times \ldots \times s_{k,n_k}$.

More aggressive converter implementations The converter in Figure 6 does not send any shared signals until it is received. This will render certain protocols inherently incompatible. One such example is shown in Figure 7(a). In this example, both `ready` and `ok` are shared control signals. Initially, P and Q are waiting for `ready` and `ok` respectively. Since the converter cannot send these signals until they are received, no progress is possible. Thus, the final state of the converter (as defined in the preceding) is not reachable from its initial state. An alternative implementation proceeds as follows. The converter sends off `ready` to P. Since this is a speculatively generated shared signal, the converter will remember this as a deficit and will not reach its final state till this deficit is erased by actually receiving this signal from Q. If it does not receive this signal from Q then the component Q will get stuck in this node. We show a protocol converter for Figure 7(a) employing speculative sending of signals in Figure 7(b); the final state is marked as f, and all speculative sends by the converter are marked in italics. Note that there exists no path from the initial to the final state where a signal sent speculatively is not actually generated.

Inherent Incompatibility detection As mentioned earlier, the converter can not speculatively send data signals. Thus, in Figure 7(a), if `ready` and `ok` are replaced by two data signals `data1` and `data2` we cannot synthesize a converter; in this case we deem the protocols to be *inherently incompatible*. In other words, if in the process of synthesizing the converter as a finite state machine, at any state if the only moves available are to speculatively send data signals then we declare incompatibility and abort. We can however perform the following static check to detect such inherent incompatibility.

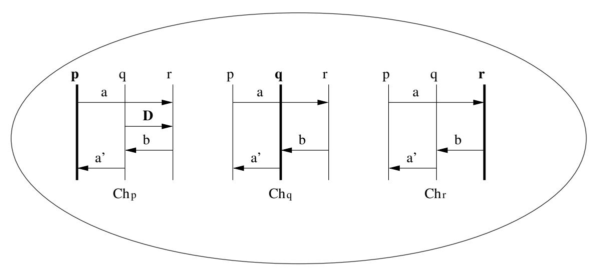


Figure 8: An example of inherently incompatible protocols

Let $\{p_1, p_2, \ldots, p_k\}$ be the agents of the node n in an HMSC with Ch_i being the MSC associated with the p_i-view at node n. Let the *data projection* of a lifeline in any one of the MSCs be the sequence of events associated with this lifeline that is obtained by erasing all occurrences of control signals. For instance, for the example shown in Figure 8, the data projection of q's lifeline in Ch_p is $\langle q!r, D\rangle$ (a sequence of length one) whereas the data projection of r's lifeline in Ch_q is ϵ, the null sequence. We will say that p_i-views ($1 \leq i \leq k$) in node n are *compatible* iff for each $i, j, m \in \{1, 2, \ldots, k\}$ the data projection of the lifeline of p_m in Ch_i is identical to the data projection of the lifeline of p_m in Ch_j. We show an example of an incompatible node in Figure 8. This node is incompatible because the data projection of q's lifeline in Ch_p (namely $\langle q!r, D\rangle$), is not identical to the data projection of q's lifeline in Ch_q (and Ch_r), namely the null sequence ϵ. As this example shows, our notion of incompatibility is a strong one. What p believes to be true about the data flow between q and r must agree with what q and r believe to be the case. This is because the computations and the communications carried out by the component p internally later on may well be influenced by its belief concerning data transfers that may or may not have taken part between two third-party agents q and r in the current node of the HMSC.

Restrictions Our current work does not address 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 can chop/merge/rotate data packets to satisfy the protocols of both the processes as discussed in [17].

4.2. Developing multi-threaded converters

In the previous subsection, 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 converter automatically generated by our toolkit.

Why do the converter threads need to communicate ? Recall that any shared signal s is received from a process p by the converter and sent to another process q. If s is a data signal (which means it cannot be sent speculatively), the converter thread servicing process q needs to know whether the converter thread servicing process p already received the signal. If yes, then the converter thread servicing process q can send the signal s to process q. Clearly only shared signals will be exchanged between converter threads via message buffers. In addition, any converter thread consumes input signals and generates output signals without communicating with other converter threads.

We develop a multi-threaded protocol converter for the HMSC specification of Figure 4 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 of Figure 4. According to the P-view, T_P executes the sequence of events ?`req`, ?`data`, !`ack` and T_Q executes the sequence of events ?`ready`, !`data`, !`finish`, ?`ack`. Any interleaving of T_P and T_Q is allowed. The task involved within the converter for the various events mentioned in the preceding are different. Since `req` and `ready` are input signals their receipt only requires the converter to wait until the signals actually arrive. `finish` is an output signal so it is simply sent off by converter thread T_Q. Since `data` is a shared data signal, when T_P executes `?data` it appends `data` to the buffer $q_{P,Q}$. When T_P executes `!data` it first checks whether `data` appears in $q_{P,Q}$; if so, it removes `data` from $q_{P,Q}$ and sends off `data` to process Q. Since `ack` is a shared signal its handling is similar to `data`, except that it can also be sent speculatively.

4.3. Converter for HMSC specifications

We now extend our converter generation technique 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, we can view the converter at each node as a multi-threaded system. 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 $1 \le j \le k$) in node n_i to the beginning of thread T_j in node n_{i+1}. 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 a HMSC has multiple successors (say $n \to n_1$ and $n \to n_2$), 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 [19]). 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 section 3). 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 it is, 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.

5. Implementation

In this section, 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 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 [10, 18].

Our tool takes as input the HMSC description of interactions among the components. This description comes in two parts: the graph structure of the HMSC and the component views (or MSCs) inside each node. 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 9.

The first 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 text file. The second part of the input, 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

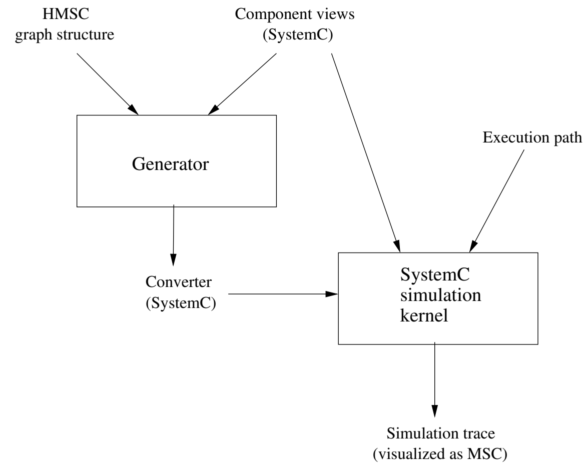


Figure 9: Overall structure of the implementation

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 10 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
 }
```

The generated converter is a SystemC module too. The converter is multi-threaded. As discussed in Section 4.3, 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 10.

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

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 [11]). Incorporating these mechanisms in our converter generation toolkit is a topic of future work.

6. Examples

In this section, 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 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 do not model the bus-controller as a separate process. Instead, we synthesize the bus controller as the protocol converter which enables communication between masters and slaves. Note that we do not impose any constraints on the synthesized bus-controller beyond enabling masters/slaves to execute their individual projected views. In contrast, the work of [16] allows specification of additional temporal properties of the converter to be synthesized via a "specification automaton".

We have modeled several features from existing SoC bus protocols which enable high speed data transfer *e.g.* split transactions. Many of these features are common in SoC bus protocols such as AMBA [2], CoreConnect [5] etc.

The first example is shown in Figure 10. 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 10 indicates data transfer (*i.e.* it is a data signal). Note though only one master has been modeled, the effect of other masters is implicitly captured via the `grant` and `nogrant` signals.

Another example modeling two masters with different priorities (assigned statically) is shown at Figure 11. 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 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. The details of these examples are not shown here due to space limitations.

Finally, we present the simulation results of the generated converter for the two simple examples we presented in this section. Figure 12(a) shows the simulation run of the synthesized converter for the HMSC specification in Figure 10; 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`).

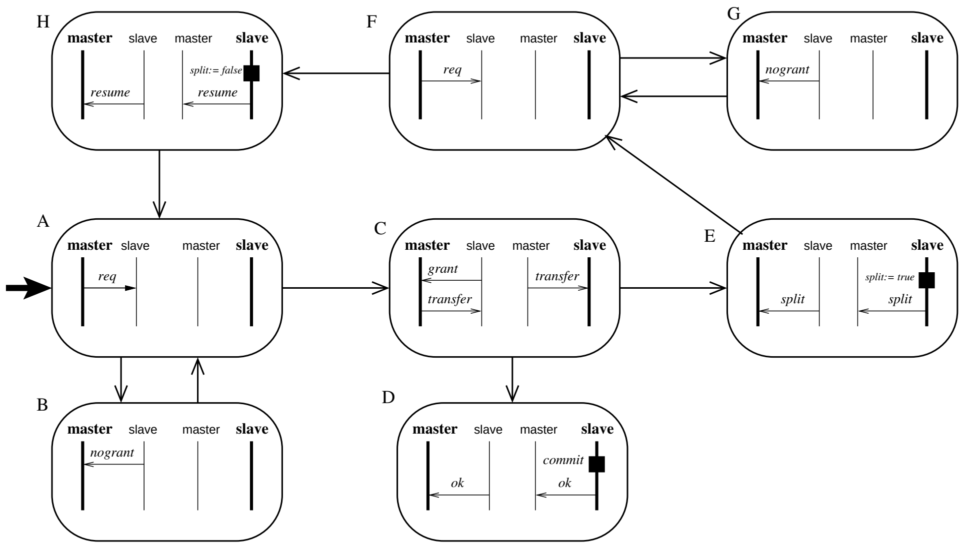


Figure 10: Example 1: Split transfers

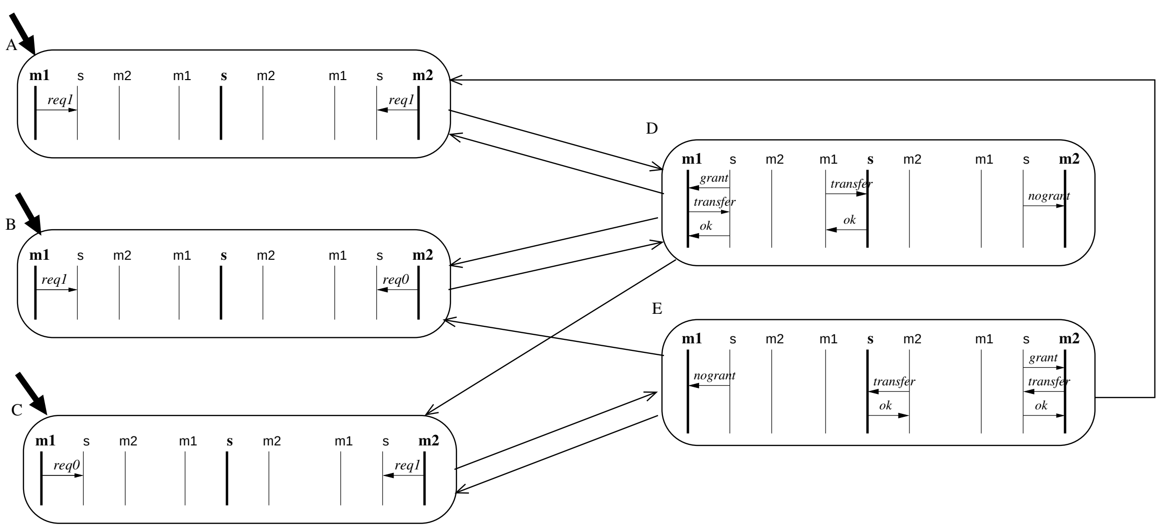
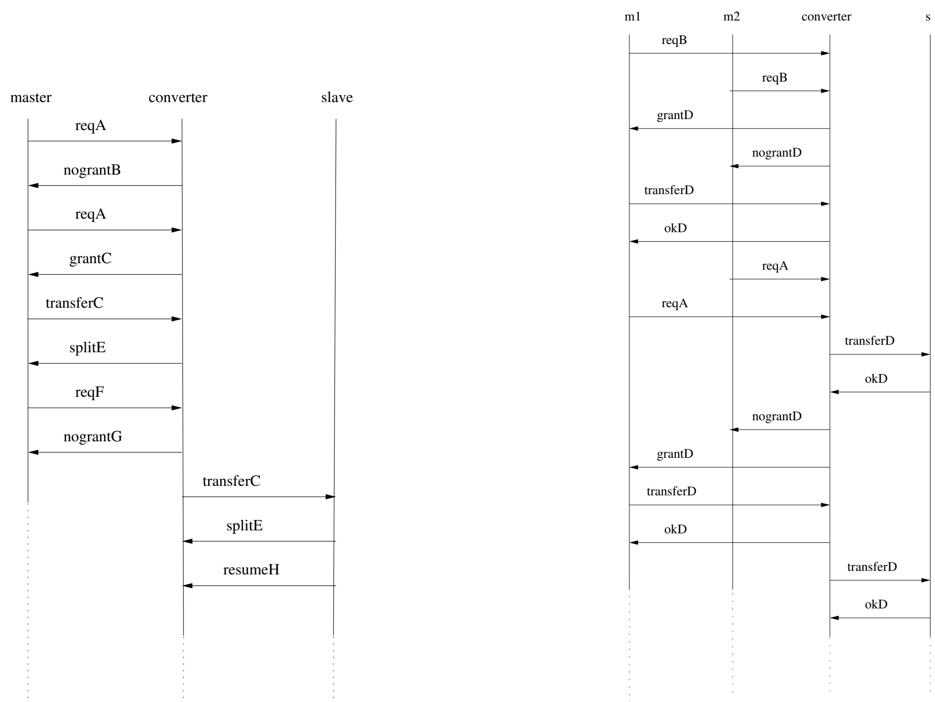


Figure 11: Example 2: Bus transfers guided by master priorities



(a) (b)

Figure 12: Simulation run of synthesized converter for (a) Figure 10 with input path `ABACEFGFHABACD` and (b) Figure 11 with input path `BDADCEAD`

- 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 12(b) shows the simulation run of the synthesized converter for the HMSC specification in Figure 11; the input path through the HMSC which was used to drive this simulation is `BDADCEAD`.

We present the 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 12(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).

7. Discussion

In this paper, we have investigated the problem of automatically generating converters which enable communication among embedded system components with 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.

In terms of future work, there exist various opportunities for extending our converter generator's capabilities. One direction will be to handle data formatting where the converter can combine/rotate data packets, separate the packet header etc. Another direction will be to include specific semantic information concerning resource sharing and use this to restrict the speculative generation of control signals. Yet another line of study will be to specialize the converter generation technique to handle software/hardware and hardware/hardware interfaces. Finally, we need to incorporate the means for imposing additional behavioral constraints on the converter; beyond the basic one handled in this paper, namely, protocol compatibility.

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.

Acknowledgments

We would like to thank the anonymous reviewers for their comments. This work was partially supported by the A*STAR research grant 022 106 0042 under the Embedded and Hybrid Systems (EHS) Programme.

References

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

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

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

[4] A. Chakrabarti, L. de Alfaro, T. Henzinger, and F. Mang. Synchronous and bidirectional component interfaces. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2002.

[5] Coreconnect. $CoreConnect^{TM}$ bus architecture. `http://www-306.ibm.com/chips/products/coreconnect/`.

[6] L. de Alfaro and T. Henzinger. Interface automata. In *Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC-FSE)*, 2001.

[7] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *International Workshop on Embedded Software (EMSOFT)*, 2001.

[8] L. de Alfaro, T. Henzinger, and M. Stoelinga. Timed interfaces. In *International Workshop on Embedded Software (EMSOFT)*, 2002.

[9] D. Gajski, J. Zhu, and R. Domer. *Specification Language: SpecC: Design Methodology*. Kluwer, 1997.

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

[11] 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.

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

[13] D. Harel and P. 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.

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

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

[16] R. Passerone, L. de Alfaro, T. Henzinger, and A. Sangiovanni-Vincentelli. Convertibility verification and converter synthesis: Two faces of the same coin. In *Intl. Conf. on Computer Aided Design (ICCAD)*, 2002.

[17] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Design Automation Conference (DAC)*, 1998.

[18] SystemC website. `http://www.systemc.org`.

[19] 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.

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

[21] V. Zvereva. Converter between incompatible protocols specified using MSC diagrams. Technical report, National Univ. of Singapore, 2004. `http://www.comp.nus.edu.sg/~abhik/pdf/Converter.pdf`.