An Executable Specification Language Based on Message Sequence Charts

Abhik Roychoudhury and P.S. Thiagarajan

School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543.
{abhik,thiagu}@comp.nus.edu.sg

Abstract. Message Sequence Charts (MSCs) are an appealing visual formalism that play a useful role in the early design stages of reactive systems such as telecommunication protocols. They also constitute one of the behavioral diagram types in the UML framework [4]. MSCs are usually intended to capture system requirements. However there is no standard relationship between such requirements and an executable specification . Here we deploy MSCs instead as refinements of actions at the executable level by formulating a state-based model called Cyclic Transaction Processes. We provide a transition system semantics for the CTP model as also a detailed example to illustrate its modeling and behavioral features.

1 Introduction

Message Sequence Charts (MSCs) are an appealing visual formalism that play a useful role in the early design stages of reactive systems [22]. They also constitute one of the behavioral diagram types in the UML framework [7]. MSCs are usually deployed to capture system requirements. A variety of choices exist as to what ought to be the relationship between such requirements and an executable specification of the system, say, in the form of a statechart [8]. Two extreme possibilities are:

- The requirements constitute a *necessarily incomplete* testbench that the executable specification must satisfy.
- The requirements constitute a complete behavioral description and hence the executable specification must exhibit *exactly* the set of scenarios (MSCs) captured as requirements.

In the first case, the MSCs play a rather weak role and in the second instance one has to usually synthesize an executable specification from the requirements which is a difficult problem even in abstract settings [16, 15, 10, 13]. An alternative approach that has been advocated -and which we follow here- by Harel and his co-workers [6, 9, 12, 11] is to develop executable specification mechanisms called Live Sequence Charts (LSCs) that are *directly* based on MSCs.

The key idea here is that LSCs can be used to describe scenario-based liveness and safety properties : for instance, interactions which *must* occur provided some pre-condition holds and such charts are referred to as universal charts. An executable specification could then consist of a collection of universal charts (and existential charts), namely, all possible executions that do not violate the specification. This is the so-called Play-in/Play-out approach developed in the papers cited above. Even though such a description is complete and executable, it does not contain explicit control flow information. This raises questions about the efficiency of the execution engine, lack of concurrency and the ease of code synthesis from the executable specification. Our specification formalism aims address these problems.

We illustrate the main features of our MSC based executable specification by formulating the model called Cyclic Transaction Processes (CTP). The CTP model consists of a set of processes where each process represents a system component. Each process p repeatedly executes a *cyclic* pattern of transaction schemes in each of which p is a participating process. A *transaction scheme* is the unit of interaction between the processes. Naturally, different transaction schemes along the cycle could involve different sets of other processes and more than two processes may be involved in a transaction scheme. The restriction that the control flow of a single process must be cyclic makes the analysis of the model tractable. It also suffices to capture the illustrative example discussed in section 2. In our subsequent work, we have weakened this cyclicity restriction to allow for more generous control flow for the component processes [21]. In the present paper we have not included features to model the interactions with the environment but they can be easily added as shown in [21].

A transaction scheme consists of a collection of *transactions* where a transaction is modeled as an MSC. An execution of a transaction scheme will choose one of its transactions depending on the current values of the local variables of the participating processes. For instance, consider a co-processor P_m trying to transfer data to a co-processor P_s via a bus B. In addition to the three processes corresponding to these components, assume there is a master interface process I_m to mediate between P_m and the bus B and a slave interface process I_s to mediate between the bus and P_s. One possible transaction scheme in this setting would model an interaction where P_m queries I_m as to whether it can buffer data for eventual transfer to P_s via the bus. This transaction scheme will consist of (at least) two transactions ; one in which P_m requests and receives a positive response from I_m to transfer data to the buffer of I_m. This will be followed by a series of data transfer operations. In the second transaction, the response from I_m is negative because its buffer is currently full. This will terminate the transaction and there will be no data transfer.

Related Work Turning now to related work, as mentioned above, the notion of Live Sequence Charts (LSCs) [6] has been an important source of inspiration for the present work. As pointed out in [6], a variety of MSC-based mechanisms for capturing requirements are rather weak. Loosely speaking, using these mechanisms, one can merely say that a scenario *may* arise during a system execution.

However as the design moves closer to an implementation, one would also like to specify that under a given condition a particular scenario *must* arise. The LSC formalism is designed to fill this gap. Using LSC terminology, a transaction scheme may be viewed as consisting of a set of *universal* charts guarded by a total set of mutually exclusive predicates. However our model differs from LSCs in that we explicitly fix the *distributed* control flow. As a result our operational semantics -even though given in terms of conventional transition systems here- can capture concurrency whereas in the play-in play-out approach of [12], the play engine has to monitor against the violation of any universal chart in every step and hence is inherently sequential. We however envision the key logical aspects of LSCs to play a role in the formal verification of executable CTP specifications. There is a wealth of literature available concerning MSCs as evidenced in [2, 13, 14, 17–20]. Here we shall only be using the the basic definition of MSCs. We do however expect parts of the rich theory cited above to play a useful role in the analysis of CTPs.

Organization In section 2, we provide a detailed example specification capturing the data communication between two co-processors via a bus running the AMBA protocol (which is widely used in System-On-Chip designs). In the section 3, we formally describe the CTP model and provide the operational semantics for a restricted class of CTPs called *anchored* CTPs. Finally, in section 4, we sketch the issues in formal verification of CTP specifications using a CTL-like logic that can incorporate important aspects of the LSC formalism.

2 An Interface Example

Our Cyclic Transaction Processes (CTP) model consists of a finite family of cyclic processes that interact via *transaction schemes*. Each process (agent) repeatedly executes a fixed cycle of transaction schemes. A transaction scheme consists of a guarded choice between a set of *transactions* where a transaction is modeled as an MSC. The choice of a transaction in a particular execution of any transaction scheme, is a distributed choice in general.

In this section, we present an example to show the use of the CTP as a specification language. In particular, we model the data communication between two components via a bus. We call the originator of the data communication the *master* and the receiver of the communication the *slave*. Our model consists of five processes executing in parallel: the master component (called P_m), interface of the master component (called I_m), the bus controller (called BC), interface of the slave component (called I_s) and the slave component (called P_s). The master and slave components (P_m and P_s) are often processors or co-processors. The architecture of interaction between the components is shown in Figure 1. The Petri net representation shown in this figure is used here mainly for illustrative purposes. The circles stand for the local control states of the processes and the boxes represent the transaction schemes. In the diagram, the constituent guarded transactions of the various transaction schemes have not been shown.

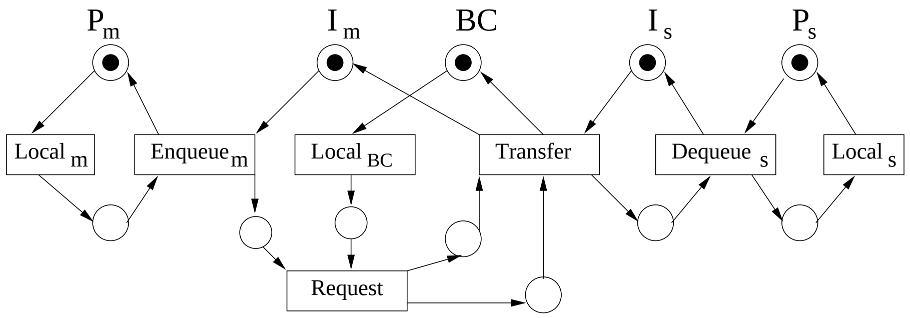

Fig. 1. CTP model of interfaces between two embedded co-processors

To develop our example, we fix: (1) a specific bus protocol, (2) storage capabilities of the interfaces, I_m and I_s (3) interaction between the components and interfaces. We choose the popular AMBA bus protocol used in ARM system-on-chip designs [3]. We assume that each interface contains a bounded queue to hold data in transit. The interaction between a component and its interface then involves enqueueing and dequeuing these queues. In particular, our choice of the component-interface protocol is drawn from the interface modules developed in the European COSY project [5].

We now proceed to model each of the processes. Each process is *cyclic*, and repeats a sequence of *transaction schemes*. A sequence of two transaction schemes T and T' is denoted as $T \circ T'$. The five processes in our example can be described by simple recursive equations as follows. An equivalent visual description of these processes appears in Figure 1.

$$\begin{aligned}
&P_m = Local_m \circ Enqueue_m \circ P_m\\
&I_m = Enqueue_m \circ Request \circ Transfer \circ I_m\\
&BC = Local_{BC} \circ Request \circ Transfer \circ BC\\
&I_s = Transfer \circ Dequeue_s \circ I_s\\
&P_s = Dequeue_s \circ Local_s \circ P_s
\end{aligned}$$

The transaction schemes $Local_m$, $Local_{BC}$ and $Local_s$ have only one participating process: namely P_m, BC and P_s respectively. They represent internal computations of these processes and we do not describe them here. $Enqueue_m$ involves enqueueing of data by the master process P_m into the queue of the master interface I_m. Similarly, $Dequeue_s$ denotes the dequeuing of data from the queue of the slave interface I_s by slave processor P_s. The scheme *Request* denotes request for bus access by the master to the bus controller, and subsequent granting of bus access (if any). Finally, the scheme *Transfer* denotes the transfer of data from master interface I_m into slave interface I_s over the bus.

We now describe the transaction scheme $Transfer$. Due to space limitations, the descriptions of the other transaction schemes, which are simpler, are not presented. Note that $agents(Transfer) = \{I_m, I_s, BC\}$. Conditions on the local variables of each of these processes are used to decide which chart of $Transfer$ is executed in a particular execution. We will freely use values of these local variables in our charts. The events in the charts pass these values between variables of different processes, thereby modeling data transfer.

Local Variables We present the local variables of the processes I_m, I_s and BC in Figure 2. We wish to note that $maxwait$ denotes a predefined fixed positive constant, and $\mathcal{D}$ denotes the data type of the data being transmitted from master component P_m to slave component P_s. Furthermore, $Addr$ denotes the range of addresses manipulated by I_m and I_s.

Process	Local Variables
I_m	mq : Queue of $(Addr, \mathcal{D})$
	$data\_sent, wait\_data$: $\mathcal{D}$
	$wait\_addr$: $Addr$
	$grant_m$: boolean
I_s	sq : Queue of $(Addr, \mathcal{D})$
	$addr\_rcvd$: $Addr$
	$waitcnt$: $0 \ldots maxwait$
BC	$gnt_m, split_m$: boolean

Fig. 2. Local Variables in the Interface Example

The master and slave interfaces I_m and I_s each contain a queue mq and sq. The master queue mq receives data from P_m and passes it to the slave interface I_s. The slave queue sq receives data from master interface I_m and passes it to the slave component P_s. The transfer of data between the master and slave interfaces is over a bus, and is thus dictated by the bus protocol. In this case, we consider the AMBA bus protocol which has the following features. This will clarify the need for the various local variables.

Bus access protocol Each transfer is preceded by a grant of bus access by the bus controller to a master. This information is stored by the bus controller BC in the boolean variable gnt_m. Its value is communicated to I_m in the *Request* transaction scheme (not shown here) when I_m requests for bus access. I_m stores this information in $grant_m$. Thus there is clear relationship between $I_m.grant_m$ and $BC.gnt_m$. Similar relationships exist between other local variables of different processes owing to the flow of values via messages.

Pipelined Transfer Multiple transfers from I_m to I_s are *pipelined*. For example suppose I_m wants to transfer (a_1, d_1),(a_2, d_2), (a_3, d_3) to I_s. This is a request to write d_1 to address a_1, d_2 to address a_2 and d_3 to address a_3. The transfer over the address and data lines proceeds as follows:

Clock cycle:	1	2	3	4
Address :	a_1	a_2	a_3	-
Data :	-	d_1	d_2	d_3

Since in every cycle, the data of the previous cycle's address is transmitted, this needs to be remembered. This information is stored in the local variable *data_sent* of I_m. Similarly, on the slave interface side, the address received in previous cycle is stored in the variable *addr_rcvd* of process I_s.

Transfer with wait cycles The slave interface I_s may not be ready to write data in every cycle *e.g.* the slave queue sq may be full. This results in insertion of "*wait cycles*". The number of such wait cycles is stored in the local variable *waitcnt*. In the presence of wait cycles, the transfer can be as follows:

Clock cycle:	1	2	3	4	5	6
Address :	a_1	a_2	a_2	a_2	a_3	-
Data :	-	d_1	d_1	d_1	d_2	d_3

Here, d_1 is transfered after two wait cycles. During these wait cycles, the master interface needs to keep on transmitting a_2 as address and d_1 as data; otherwise the correspondence between address and data is lost. Hence the need for the local variables *wait_addr* and *wait_data* in process I_m.

Split Transfer If the number of wait cycles equals a threshold $maxwait$, the slave interface I_s informs the bus controller BC that it is currently unable to service the master interface I_m. The bus controller BC then records that I_m is suspended by setting $split_m$ (which is reset later when I_s is able to serve I_m).

Sequence Charts The transaction scheme $Transfer$ is collection of MSCs, one for each of the following mutually exclusive conditions.[1]

1. $\neg grant_m \vee (empty(mq) \wedge waitcnt = 0) \vee (split_m \wedge full(sq))$
2. $grant_m \wedge \neg split_m \wedge \neg empty(mq) \wedge \neg full(sq) \wedge waitcnt = 0$
3. $grant_m \wedge \neg split_m \wedge \neg empty(mq) \wedge full(sq) \wedge waitcnt = 0$
4. $grant_m \wedge \neg split_m \wedge \neg full(sq) \wedge waitcnt > 0$
5. $grant_m \wedge \neg split_m \wedge full(sq) \wedge waitcnt > 0 \wedge waitcnt < maxwait$
6. $grant_m \wedge \neg split_m \wedge full(sq) \wedge waitcnt = maxwait$
7. $grant_m \wedge split_m \wedge \neg full(sq) \wedge waitcnt = maxwait$

In case 1, either the bus is busy ($\neg grant_m$ holds) or the master queue mq is empty and $waitcnt = 0$ (*i.e.* new data needs to be dequeued from mq which is empty), or the data transfer from I_m has been split, but I_s is still not ready to input data (sq is full). In these cases, no data is transmitted, no control signals are exchanged and the chart is a no-op.

In case 2 (shown in Figure 3), the master is granted access to the bus, data is dequeued from the master queue mq, and enqueued into the slave queue sq. This corresponds to "normal" data transfer without wait cycles and split transfer. Each message is of the form Signal_name(Value), such as ADDR(a). Access to mq and sq are through the *Enqueue* and *Dequeue* methods.

The chart for case 3 is shown in Figure 4. This corresponds to the scenario where wait cycles are initiated (note that $waitcnt = 0$) for some transfer, since

[1] These guards are also total, when the relationships between the local variables of various processes are taken into account.

the queue at I_s is full. Note that the first three actions by I_m in this chart are the same as Figure 3. This illustrates the distributed decision-making performed by agents of a transaction scheme in deciding which chart is to be executed. As long as the slave interface I_s does not execute its internal actions, we cannot decide whether chart for case 2 or case 3 is being executed.

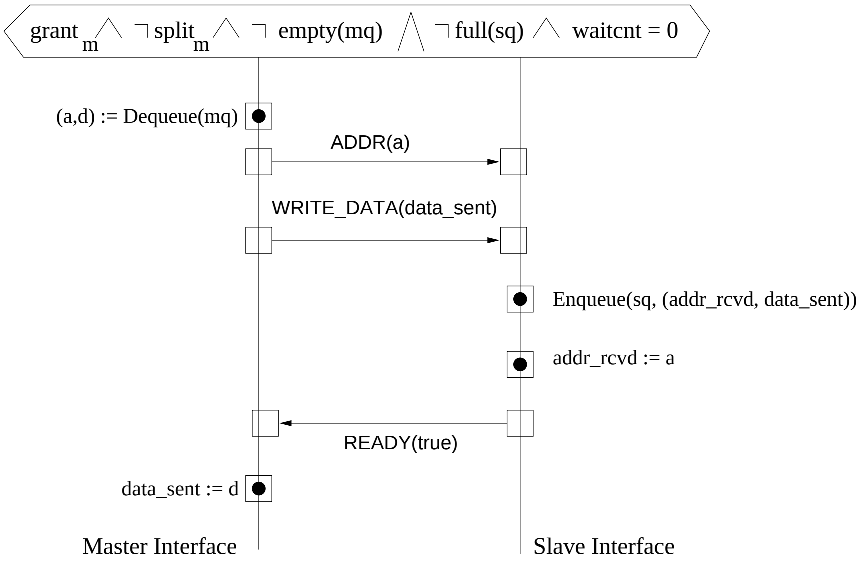

Fig. 3. Normal data transfer between master and slave interface

Case 4 corresponds to $grant_m \wedge \neg split_m \wedge \neg full(sq) \wedge waitcnt > 0$. Thus, the master has been granted bus access (since $grant_m$ holds) and is currently going through a wait cycle (since $waitcnt > 0$). The slave is however ready to input data (since $\neg full(sq)$), that is, the master need not wait any more. Thus, this scenario corresponds to the last wait cycle. The chart is shown in Figure 5.

Case 5 corresponds to $grant_m \wedge \neg split_m \wedge full(sq) \wedge waitcnt > 0 \wedge waitcnt < maxwait$. Here again the master has been granted bus access (since $grant_m$ holds) and is currently going through a wait cycle (since $waitcnt > 0$). The slave is still not ready to input data (since $full(sq)$). This scenario corresponds to a wait cycle which is not the last. The chart appears in Figure 6.

Case 6 corresponds to $grant_m \wedge \neg split_m \wedge full(sq) \wedge waitcnt = maxwait$. Here the master is going through a wait cycle, but the number of wait cycles has reached the pre-defined threshold *maxwait*. Thus, this requires the slave to initiate *split transfer* by interacting with the bus controller. The chart appears in Figure 7.

Case 7 corresponds to $grant_m \wedge split_m \wedge waitcnt = maxwait \wedge \neg full(sq)$. This means that the transfer from I_m to I_s was previously split, thus $split_m$ holds.

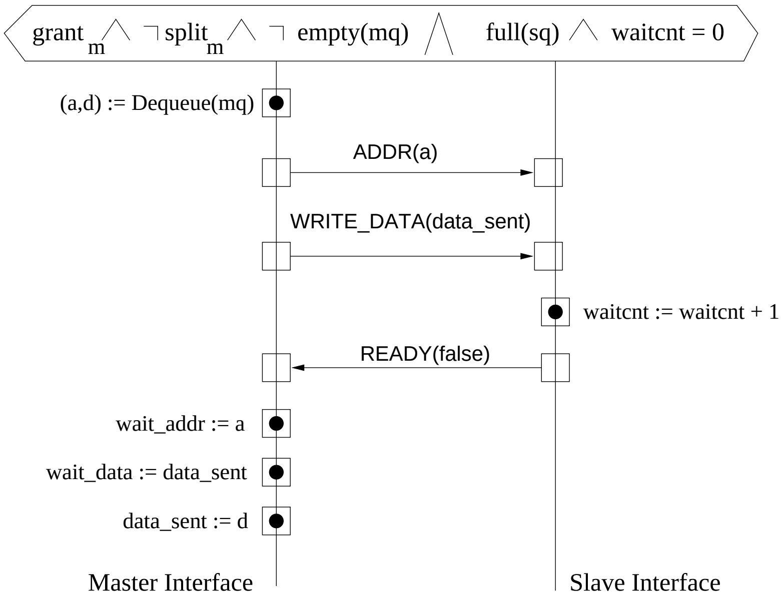

Fig. 4. Initiation of wait cycles

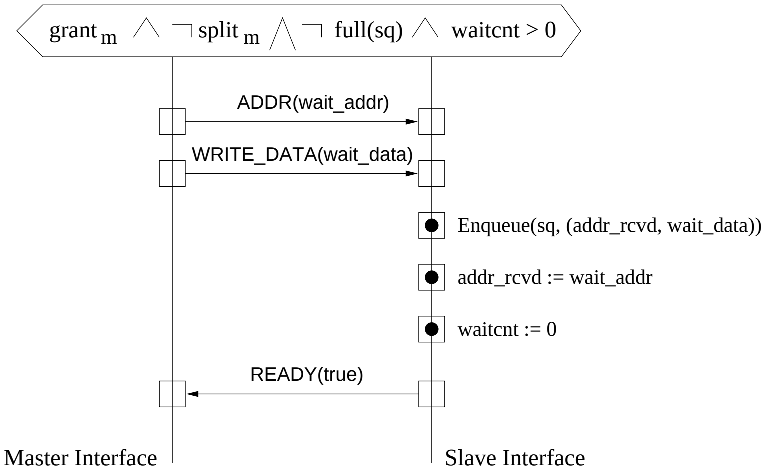

Fig. 5. The last wait cycle

However, the slave is currently ready to input data (since $\neg full(sq)$), thereby terminating the split transfer. Thus, this chart will involve exchange of SPLIT and SPLIT_ACK signals along the lines of Figure 7, and the resetting of *waitcnt* to zero.

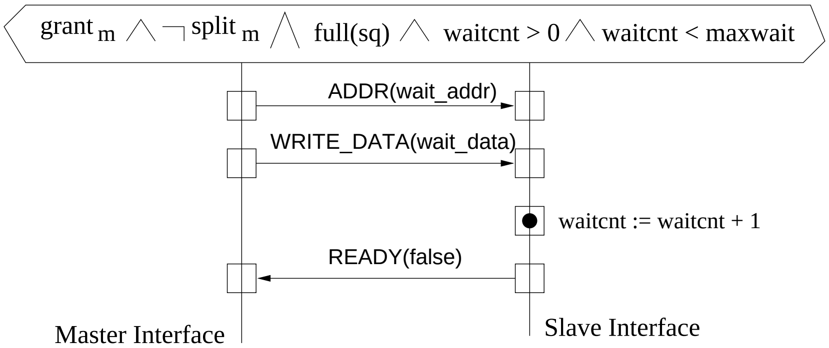

Fig. 6. A wait cycle which is not the last

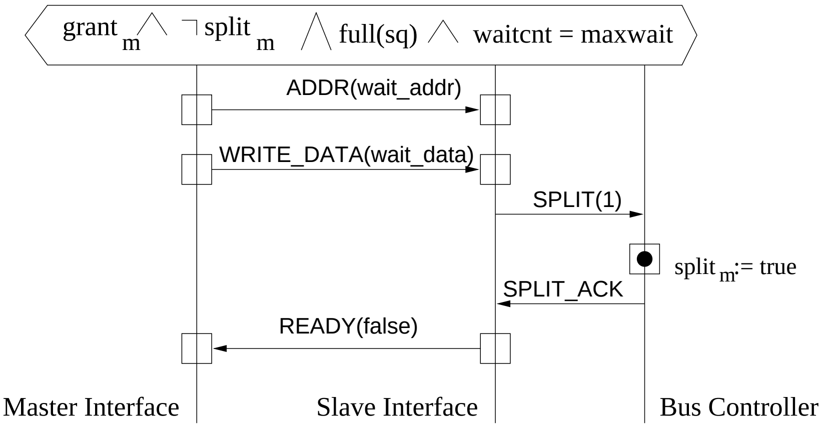

Fig. 7. Initiation of split transfer

As a matter of fact, the AMBA bus protocol is intended for interaction between multiple masters and multiple slaves. Here we have modeled only one master and one slave. However, all the features for multi-component interaction have been, in principle, captured. For example, the suspension of bus access to a master (split transfers) is to allow another master to take over bus access. In our case, even with one master we have modeled this feature via the variable $split_m$. In future, we plan to explicitly model interaction among multiple masters and slaves, that is, multiple instances of P_m and P_s.

3 Cyclic Transaction Processes

We shall build up our model, starting with the transactions. Through the rest of this section we fix a finite set of processes $\mathcal{P}$ and let p, q range over $\mathcal{P}$. We also fix a finite message alphabet M and let m, m' range over M. In our model each process will have a set of local variables whose values will be updated (and sometimes transmitted) during an execution. The values of these variables will also determine the truth-values of the predicates guarding the individual events as well the transactions. In order to simplify the formal model, we will abstract the role of these variables with the help of propositional variables. In concrete models however, as in the previous section, we will feel free to use local variables explicitly. We now fix a pair-wise disjoint family $\{Prop_p\}_{p \in \mathcal{P}}$ where each $Prop_p$ is a finite set of propositions on the local variables of process p. We also fix a finite set of internal actions Act with a, b ranging over Act.

We use Σ_p to denote the set of actions executed by p; it consists of actions of the form $(g, p!q, m, A)$, $(g, p?q, m, A)$ and (g, p, a, A). In each case, g is called the guard of the action and it is a propositional formula built out of $Prop_p$ (we identify propositionally equivalent formulae). We also require $A \subseteq Prop_p$. The communication action $(g, p!q, m, A)$ stands for p sending the message m to q when the guard g holds. As a result of this action, the set of propositions in A begin to hold. In a similar way, the communication action $(g, p?q, m, A)$ represents p receiving the message m from q with g and A interpreted as before. Finally (g, p, a, A) is an internal action of p with a being the member of Act being executed and g and A interpreted as before. We set $\Sigma = \bigcup_{p \in \mathcal{P}} \Sigma_p$ and let α, β range over Σ.

A *transaction* is finite MSC over Σ. We shall view a transaction (MSC) as a Σ-labeled poset $t = (E, \leq, \lambda)$ with $\lambda : E \to \Sigma$ satisfying the restrictions required by an MSC. We shall assume the usual formalization of MSCs (say, as in [1, 13]). We assume a simple setting where message passing via point-to-point (unbounded) FIFOs is the communication mechanism. Again, in concrete examples, as in the previous section, it is convenient to assume that the messages have an internal structure using which values can be passed between processes. In the formal model however, this will be captured with the help of the truth values of the propositional variables.

Let $t = (E, \leq, \lambda)$ be a transaction. Then $E_p = \{e \mid e \in E \text{ and } \lambda(e) \in \Sigma_p\}$. Further, $agents(t) = \{p \mid E_p \neq \emptyset\}$. We shall assume without loss of generality that $E \neq \emptyset$ so that $agents(t) \neq \emptyset$. Next suppose $X \subseteq E$. Then $\downarrow X = \{y \mid \exists x \in X \text{ with } y \leq x\}$. We shall say that X is *down-closed* in case $\downarrow X = X$. Next let $\emptyset \neq P \subseteq agents(t)$. Then the *P-fragment* of t is denoted as $t \restriction P$ and it is the Σ-labeled poset $(E^P, \leq_P, \lambda_P)$ where E^P is the largest down-closed subset of E

satisfying : if $e \in E^P$ and $e' \leq e$ then $e' \in E_p$ for some $p \in P$. Further, $\leq_P$ (λ_P) is $\leq$ (λ) restricted to $E^P \times E^P$ (E^P).

Definition 1 (Transaction Scheme) A *transaction scheme* is a finite family $T = \{G_i : t_i\}_{i=1}^k$ where :

- Each $t_i = (E_i, \leq_i, \lambda_i)$ is a transaction.
- each G_i is of the form $\bigwedge_{p \in agents(t_i)} g_i^p$ where g_i^p is a guard built out of $Prop_p$.
- If $i \neq j$ then t_i and t_j are not isomorphic.
- $\{G_i\}_{i=1}^k$ are mutually exclusive, deterministic and total in the sense :
  - Suppose $\emptyset \neq P \subseteq agents(t_i) \cap agents(t_j)$ and $\bigwedge_{p \in P}(g_i^p \wedge g_j^p)$ is satisfiable. Then the P-fragments of t_i and t_j are isomorphic.
  - $G_1 \vee G_2 \vee \ldots \vee G_k$ is a tautology.

Thus G_i is a global guard which determines when t_i is to be executed. This choice will however be made in a distributed fashion during a computation. It could be the case that at some stage during an execution of the transaction scheme T, for some $\emptyset \neq P \subseteq agents(t_i) \cap agents(t_j)$, the truth values of $\bigcup_{p \in P} Prop_p$ are such that $\bigwedge_{p \in P}(g_i^p \wedge g_j^p)$ is satisfiable. Hence at this stage we still will not know whether t_i or t_j (or some other t_l) will eventually run to completion. In any case, we require that the partial executions relative to t_i and t_j must be indistinguishable. In this sense the transactions are executed in a deterministic fashion. The above definition captures the intuition that *any particular execution of a transaction scheme $T = \{G_i : t_i\}_{i=1}^k$ must execute one of the constituent transactions of* T, that is, one of $t_1, \ldots, t_k$.

It is worth remarking here that a transaction scheme involving just one agent and only internal actions can be used to model interactions with the environment. These special transaction schemes can be allowed to execute non-deterministically so as to capture the lack of control over the inputs supplied by the environment.

In the following, we shall denote by $agents(T)$ the set of processes given by $agents(T) = \bigcup_{1 \leq i \leq k} agents(t_i)$ where $T = \{G_i : t_i\}_{i=1}^k$ is a transaction scheme. We now define the CTP model:

Definition 2 (CTP) A CTP is a structure $\mathcal{S} = (\{S_p\}_{p \in \mathcal{P}}, \mathcal{T}, \theta, s_{in}, V_{in})$ where:

- Each S_p is a finite set of p-local states, the local control states of p.
- $\mathcal{T}$ is a finite set of transactions schemes.
- θ is a distribution function which assigns to each $T \in \mathcal{T}$ a pair $(pre_T, post_T)$ with $pre_T, post_T : agents(T) \to \bigcup_{p \in \mathcal{P}} S_p$ such that $pre_T(p), post_T(p) \in S_p$ for each $p \in agents(T)$.
- $s_{in} \in \prod_{p \in \mathcal{P}} S_p$ is the global initial control state.
- V_{in} is the initial valuation function which assigns to each p a subset of $Prop_p$.
- To capture the cyclicity property, we let $\mathcal{T}_p = \{T \mid p \in agents(T)\}$ for each p. We also define $R_p = \{(s, T, s') \mid T \in \mathcal{T}_p$ and $s = pre_T(p)$ and $s' = post_T(p)\}$. We now require that, for each p, the sets S_p and $\mathcal{T}_p$ can be indexed as $S_p = \{s_1, s_2, \ldots, s_{n_p}\}$ and $\mathcal{T}_p = \{T_1, T_2, \ldots, T_{n_p}\}$ so that $R_p = \{(s_i, T_i, s_{i+1}) \mid 1 \leq i < n_p\} \cup \{(s_{n_p}, T_{n_p}, s_1)\}$.

It is the cyclicity requirement which makes the CTP model tractable. Under this regime, each agent p will repeatedly (attempt to) execute $T_1.T_2.\ldots T_{n_p}$ with the corresponding element in S_p signaling in which portion of the cycle p is currently residing. The function pre_T specifies the local control state that each $p \in agents(T)$ must be in (given by $pre_T(p)$) for it to execute actions belonging to Σ_p within the transactions appearing in T. Similarly $post_T$ specifies the resulting control state for each $p \in agents(T)$ when p has completed all its actions relative to T. In our model, this initiating and terminating of a transaction scheme will in general be *asynchronously* carried out by the participating agents. In other words, we shall consider an interleaved asynchronous semantics here. We can however, use it to model synchronously executing systems; a set of actions being executed in a single step will be then captured by all possible interleavings of this set of actions in the present setting.

Operational Semantics We now turn to the operational semantics of the CTP model. In order to bring out the main ideas clearly we shall define the semantics only for a restricted subclass called *anchored* CTPs. The example treated in the previous section is an example of anchored CTP. To define anchored CTPs, let $t = (E, \leq, \lambda)$ be a transaction. Then $p \in agents(t)$ is called the *master process* of t if there exist e_{in} and e_{fin} in E_p such that e_{in} is the smallest event and e_{fin} is the greatest event in E under $\leq$. Thus the master process initiates and terminates the whole transaction. The transaction scheme $T = \{G_i : t_i\}_{i=1}^{k}$ is said to be *anchored* in case there exists $p \in agents(T)$ such that p is the master process of t_i for every $i \in \{1, 2, \ldots, k\}$. Finally the CTP $\mathcal{S}$ is said to be anchored in case all its transactions schemes are anchored. An important consequence of this restriction is that at most one instance of a transaction scheme will be active during an execution and as a result, the state space turn out to be finite.

We now fix an anchored CTP $\mathcal{S} = (\{S_p\}_{p\in\mathcal{P}}, \mathcal{T}, \theta, s_{in}, V_{in})$. Our operational semantics associates a labeled transition system with $\mathcal{S}$ whose label set is $\mathcal{T} \times \Sigma$. The states of this transition system are called configurations. Each configuration consists of four components : a global control state, a valuation function, the states of the communication channels attached to the transaction schemes and a progress table. This last component keeps track of the currently active transaction schemes. Within a scheme, the progress table keeps track of the potential transactions being executed where the final identity of the transaction will be determined only when the truth values of the relevant guards become available in the vicinity of the transaction scheme.

Let $T = \{G_i : t_i\}_{i=1}^{k}$ be a transaction scheme with $t_i = (E_i, \leq_i, \lambda_i)$. A *progress function* for T assigns to each i the symbol $\#$ or a *down-closed* subset of E_i such that the following conditions are satisfied :

- $f_T(i) \neq \#$ for some i.
- If $f_T(i) \neq \#$ then $f_T(i) \subset E_i$.
- Suppose $f_T(i) \neq \#$, $f_T(j) \neq \#$ and $\preceq_i$ ($\preceq_j$) is $\leq_i$ ($\leq_j$) restricted to $f_T(i) \times f_T(i)$ ($f_T(j) \times f_T(j)$). Further, ν_i (ν_j) is λ_i (λ_j) restricted to $f_T(i)$ ($f_T(j)$). Then $(f_T(i), \preceq_i, \nu_i)$ and $(f_T(j), \preceq_j, \nu_j)$ are isomorphic.

$f_T(i) = \#$ denotes that t_i is ruled out for the current execution. If both t_i and t_j are considered to be currently active then we cannot distinguish between t_i and t_j in terms of the actions that have been executed so far. If $f_T(i) \neq \#$ and $f_T(i) \neq \emptyset$ for some i, we will say that T is active under f.

A *progress table* for $\mathcal{T}$ is a family $\{f_T\}_{T\in\mathcal{T}}$ where each f_T is a progress function for T. The active transaction under a progress table are the transactions that are active under the corresponding progress functions.

Channel states will capture the contents of the communication channels. Let $T \in \mathcal{T}$. Then $Ch_T = \{(p,q) \mid p \neq q \text{ and } p,q \in agents(T)\}$. A T-channel state is a function $\xi_T : Ch_T \to M^\star$.

We define a *configuration* of $\mathcal{S}$ as a quadruple $(s, V, \{\xi_T\}_{T\in\mathcal{T}}, \{f_T\}_{T\in\mathcal{T}})$ s.t.

- $s \in \prod_{p\in\mathcal{P}} S_p$ is a global control state.
- V is a valuation that assigns a subset of $Prop_p$ to each p.
- ξ_T is a T-channel state for each T.
- $\{f_T\}_{T\in\mathcal{T}}$ is a progress table for $\mathcal{T}$.

The initial configuration is $c_{in} = (s_{in}, V_{in}, \{\xi_T^{in}\}_{T\in\mathcal{T}}, \{f_T^{in}\}_{T\in\mathcal{T}})$ whose first two components are as in $\mathcal{S}$. Further, $\xi_T^{in}(ch) = \epsilon$ for every T and every $ch \in Ch_T$. Finally, for each $T \in \mathcal{T}$ with $T = \{G_i : t_i\}_{i=1}^k$, $f_T^{in}(i) = \#$ if there exists $p \in agents(T)$ such that $s_{in}(p) = pre_T(p)$ and $V_{in}(p)$ does not satisfy g_i^p; otherwise $f_T(i) = \emptyset$ (recall that $G_i = \bigwedge_{p\in\mathcal{P}} g_i^p$).

We define the notion of a transition label being enabled. The set of transition labels is $L = \mathcal{T} \times \Sigma$. Let $c = (s, V, \{\xi_{\widehat{T}}\}_{\widehat{T}\in\mathcal{T}}, \{f_{\widehat{T}}\}_{\widehat{T}\in\mathcal{T}})$ be a configuration and $(T,\alpha) \in L$. To define the notion of (T,α) being *enabled* at c, consider $T = \{G_i : t_i\}_{i=1}^k$ with $t_i = (E_i, \leq_i, \lambda_i)$ for each i. We first construct $J \subseteq \{1, 2, \ldots, k\}$ and the family of events $\{e_j\}_{j\in J}$ via : $j \in J$ iff $f_T(j) \neq \#$ and there exists $e_j \in E_j - f_T(j)$ such that $f_T(j) \cup \{e_j\}$ is down-closed and $\lambda_j(e_j) = \alpha$. Now (T,α) is enabled at c iff the following are satisfied (we assume $\alpha \in \Sigma_p$):

- $J \neq \emptyset$ (events which must precede α have completed in a transaction of T)
- $V(p)$ satisfies g where g is the guard of α (the first component of α).
- $s(p) = pre_T(p)$ ((T,α) could be the first action to be executed by T).
- If α is of the form $(g, p?q, m, A)$ then $\xi_T((p,q))$ is of the form $\sigma.m$ with $\sigma \in M^\star$ (message m is at the head of the FIFO (p,q)).

We define $\longrightarrow$, the *transition relation*. Consider transition label $(T,\alpha) \in L$ and configuration $c = (s, V, \{\xi_{\widehat{T}}\}_{\widehat{T}\in\mathcal{T}}, \{f_{\widehat{T}}\}_{\widehat{T}\in\mathcal{T}})$. Then $c \stackrel{(T,\alpha)}{\longrightarrow} c'$ iff (T,α) is enabled at c and $c' = (s', V', \{\xi'_{\widehat{T}}\}_{\widehat{T}\in\mathcal{T}}, \{f'_{\widehat{T}}\}_{\widehat{T}\in\mathcal{T}})$ satisfies (we assume $\alpha \in \Sigma_p$):

- Let $q \in agents(T)$. Suppose $s(q) = pre_T(q)$ and $(f_T(j) \cup \{e_j\}) \cap E_j^q = E_j^q$ for every $j \in J$ (recall the index set J computed in the definition of the enabledness of (T,α) at c). Then $s'(q) = post_T(q)$. Otherwise $s'(q) = s(q)$. If $q \notin agents(T)$ then $s'(q) = s(q)$.[2]

[2] Once the transactions t_j with $j \in J$ are updated, it could turn out that some process $q \neq p$ has no more events to execute whereas, before the occurrence of (T,α) there may have been an active transaction t_l in which q still had some events to execute.

- $V'(p) = A$ where A is the last component of α. If $q \neq p$ then $V'(q) = V(q)$.
- $\xi'_T((p,q)) = m.\xi_T((p,q))$ when α is of the form $(g, p!q, m, A)$. If α is of the form $(g, p?q, m, A)$ then $\xi'_T((p,q)) = \sigma$ where $\xi_T((p,q)) = \sigma.m$; finally if α is an internal action then $\xi'_T = \xi_T$. For all other transaction schemes $\widehat{T} \neq T$ we set $\xi'_{\widehat{T}} = \xi_{\widehat{T}}$.
- To define f'_T, it is convenient to first define the predicate (PR) as follows.

$$\text{(PR) } J = \{j\} \text{ is a singleton and } f_T(j) \cup \{e_j\} = E_j{}^3$$

Now suppose (PR) is satisfied. Then $f'_T(i) = \#$ if there exists q such that $s'(q) = pre_T(q)$ and $V'(q)$ does not satisfy g_i^q (as before $T = \{G_i : t_i\}_{i=1}^k$ and $G_i = \bigwedge_{q \in \mathcal{P}} g_i^q$). Otherwise $f'_T(i) = \emptyset$.[4] In case (PR) is not satisfied, $f'_T(j) = \#$ if $j \notin J$ and $f'_T(j) = f_T(j) \cup \{e_j\}$ if $j \in J$.

For all other transaction schemes $\widehat{T} \neq T$, we define $f'_{\widehat{T}}$ as follows. Suppose $\widehat{T} \neq T$ with $\widehat{T} = \{\widehat{G}_i : \widehat{t}_i\}_{i=1}^{\widehat{k}}$. Then $f'_{\widehat{T}}(i) = \#$ if there exists q such that $pre_{\widehat{T}}(q) = s'(q)$ and $V'(q)$ does not satisfy $\widehat{g}_i^q$ (we are assuming here that $\widehat{G}_i = \bigwedge_{q \in \mathcal{P}} \widehat{g}_i^q$). Otherwise $f'_{\widehat{T}}(i) = f_{\widehat{T}}(i)$.

Finally, the *transition system* associated with $\mathcal{S}$ is $TS_{\mathcal{S}} = (C_{\mathcal{S}}, L, \Longrightarrow, c_{in})$ where $C_{\mathcal{S}}$ and $\Longrightarrow \subseteq C_{\mathcal{S}} \times L \times C_{\mathcal{S}}$ are the least sets satisfying *(i)* $c_{in} \in C_{\mathcal{S}}$, and *(ii)* if $c \in C_{\mathcal{S}} \wedge c \xrightarrow{(T,\alpha)} c'$ then $c' \in C_{\mathcal{S}}$ and $c \overset{(T,\alpha)}{\Longrightarrow} c'$. It is easy to see that $TS_{\mathcal{S}}$ is a *finite state* transition system. Consequently, interfaces specified with the help of anchored CTPs can be realized as finite state devices.

4 Analysis and Verification Issues

In section 3, we provided the operational semantics for anchored CTPs. This semantics can however be extended to handle all CTPs [21]. The main complication is that, in general, more than one instance of a transaction scheme can be active at a configuration. The number of such instances can be unbounded. The major change in the operational semantics, to handle such CTPs is in the definition of the progress table and the manner in which the table is updated during a transition. The main idea is that the entry in the table corresponding to a transaction scheme T will consist of an ordered set of progress functions $\{f_T^1, f_T^2, \ldots, f_T^m\}$ where m can grow unboundedly. The indexing will reflect the "age" of the various instances with the earlier entries deleted (upon completion) before the later ones. A CTP is *bounded* iff the associated transition system is finite under the full semantics. Anchored CTPs (presented in the last section) will be bounded under the full semantics.

[3] This signifies that t_j is the final identity of the transaction that has been chosen for execution and it has just finished.

[4] We need to prepare T for the next possible execution. Some of the transactions in T may already be ruled out by the updated valuation V' (in case $pre_T(q) = post_T(q)$). For those that are not ruled out, we reset the progress function via $f'_T(i) = \emptyset$.

It is not difficult to associate a Petri net with a CTP and hence determining if a CTP is bounded is effectively decidable. A Petri net based operational semantics of the full CTP model appears in [21]. In fact, in this work we show that even the cyclicity restriction of CTP can be weakened.

We can reason about bounded CTPs using standard temporal logics. In fact, we can even use features of the LSC specification language to describe properties of bounded CTPs. To sketch the main ideas, let $\mathcal{S}$ be a bounded CTP and $TS_{\mathcal{S}}$ over be its associated transition system. We can view $TS_{\mathcal{S}}$ as a Kripke structure with $Prop = \bigcup_{p \in \mathcal{P}} Prop_p$ as the set of atomic propositions. We can interpret formulas of the following variant of CTL over this Kripke structure with x ranging over $Prop$ and α ranging over the set of actions Σ.

$$CTPL(\mathcal{P}, Prop) ::= x \mid \sim \varphi \mid \varphi \vee \varphi' \mid \langle \alpha \rangle \varphi \mid EU(\varphi, \varphi') \mid AU(\varphi, \varphi')$$

Thus the only variation from CTL is the next (EX) operator which is now indexed by an action. The assertion $\langle \alpha \rangle \varphi$ holds at a configuration c if there exists a configuration c' and a transaction T such that $c \stackrel{(T,\alpha)}{\Longrightarrow} c'$ and φ holds at c'. The remaining operators are interpreted in the usual way.

We now encode the existential and universal charts of LSC into this logic along the lines suggested in [10]. Let $t = (E, \leq, \lambda)$ be a transaction, $lin(t)$ be the set of linearizations of the poset $(E, \leq)$ and $Lin(t) = \{\lambda(\tau) \mid \tau \in lin(t)\}$ where λ also denotes the natural extension of λ to $E^\star$. We will say that an existential chart $\langle\langle t \rangle\rangle$ is satisfied at a configuration c if there *exists* a path $c_0 \stackrel{(T_0,\alpha_0)}{\Longrightarrow} c_1 \stackrel{(T_1,\alpha_1)}{\Longrightarrow} c_2 \ldots c_n \stackrel{(T_n,\alpha_n)}{\Longrightarrow} c_{n+1}$ with $c = c_0$ (i.e finite path from c) such that:

- $\sigma \restriction \Sigma_t \in Lin(t)$ where $\sigma = \alpha_0.\alpha_1.\ldots.\alpha_n$, $\Sigma_t = \{\lambda(e) \mid e \in E\}$ and $\sigma \restriction \Sigma_t$ is obtained from σ by deleting all appearances of letters not in Σ_t.

Similarly we will say that a universal chart $[[t]]$ is satisfied at a configuration c if *every* path from c has a prefix whose Σ_t-projection is in $Lin(t)$. Both these assertions (existential and universal charts) can be expressed in CTPL. We can also encode in CTPL existential and universal charts with pre-charts or activation messages (as presented in [6, 10]). Furthermore, one can devise linear time temporal logics in a standard way for bounded CTPs.

Acknowledgments Luciano Lavagno suggested using MSCs to model component interfaces in embedded systems and this led to the formulation of the CTP model. He also provided a concrete example and valuable insights during discussions with the second author. This work was partially supported by a National University of Singapore Research Grant R-252-000-103-112.

References

1. R. Alur, G.J. Holzmann, and D.A. Peled. An analyzer for message sequence charts. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1055*, 1996.

2. R. Alur and M. Yannakakis. Model checking of message sequence charts. In *International Conference on Concurrency Theory (CONCUR), LNCS 1664*, 1999.
3. ARM Limited. *AMBA On-chip Bus Specification*, 1999.
4. G. Booch, I. Jacobsen, and J. Rumbaugh. *Unified Modeling Language for Object-oriented development.* Rational Software Corporation, 1996.
5. Codesign, Simulation and Synthesis (COSY) project. *Generic Interface Modules for PI-Bus*, 2001.
6. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.
7. B.P. Douglass. *Doing Hard Time: Developing Real-time Systems using UML, Objects, Frameworks and Patterns.* Addison-Wesley, 1999.
8. D. Harel. Statecharts: A visual formalism for complex ystems. *Science of Computer Programming*, 8, 1987.
9. D. Harel and H. Kugler. From play-in scenarios to code: An achievable dream. In *Fundamental Approaches to Software Engineering (FASE), LNCS 1783*, 2000.
10. D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. In *International Conference on Implementation and Applications of Automata (CIAA), LNCS 2088*, 2001.
11. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. Technical report, Weizmann Institute of Science, 2002.
12. D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out apprroach. Technical report, Weizmann Institute of Science, 2001.
13. J.G. Hendriksen, M. Mukund, K.N. Kumar, and P.S. Thiagarajan. Message sequence graphs and finitely generated regular MSC languages. In *International Colloquium on Automata, Languages and Programming (ICALP), LNCS 1853*, 2000.
14. J.G. Hendriksen, M. Mukund, K.N. Kumar, and P.S. Thiagarajan. Regular collections of message sequence charts. In *Mathematical Foundations of Computer Science (MFCS), LNCS 1893*, 2000.
15. I. Krueger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *International Workshop on Distributed and Parallel Embedded Systesms (DIPES)*, 1998.
16. S. Leue, L. Mehrmann, and M. Ressi. Synthesizing ROOM models from message sequence chart specifications. Technical report, University of Waterloo, Tech Report 98-06, 1998.
17. P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. In *International Colloquium on Automata, Languages and Programming (ICALP), LNCS 2076*, 2001.
18. A. Muscholl and D.A. Peled. Deciding properties of message sequence charts. In *Foundations of Software Science and Computation Structures(FoSSaCS)*, 1998.
19. A. Muscholl and D.A. Peled. Message sequence graphs and decision problems on Mazurkiewicz traces. In *Mathematical Foundations of Computer Science(MFCS), LNCS 1672*, 1999.
20. A. Muscholl and D.A. Peled. From finite state communication protocols to high-level message sequence charts. In *International Colloquium on Automata, Languages and Programming (ICALP), LNCS 2076*, 2001.
21. A. Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. Technical report, National University of Singapore, 2002.
22. Z.120. Message Sequence Charts (MSC'96), 1996.