Communicating Transaction Processes: An MSC-based Model of Computation for Reactive Embedded Systems

Abhik Roychoudhury and P.S. Thiagarajan

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

Abstract. Message Sequence Charts (MSC) have been traditionally used to depict execution scenarios in the early stages of design cycle. MSCs portray inter-object interactions. Synthesizing intra-object executable specifications from an MSC-based description is a non-trivial task. Here we present a model of computtaion called *Communicating Transaction Processes* (CTP) based on MSCs from which an executable specification can be extracted in a straightforward manner. Our model describes a network of communicating processes in which the processes interact via common action labels. Each action is a non-atomic interaction described as a guarded choice of MSCs. Thus our model achieves a separation of concerns: the high-level network of processes depicting intra-process computations and control flow, while the common non-atomic communication actions capture inter-process interaction via MSCs. We show how to extract an ordinary Petri net from a CTP model thereby leading to a standard operational semantics. We also discuss the connection of our formalism to Live Sequence Charts, an extension of MSCs which also has an executable semantics.

1 Introduction

Message Sequence Charts (MSCs) are an attractive visual formalism which are used in the early design stages of reactive systems. They portray scenarios that arise from component interactions and hence can be used to capture requirements and test cases. MSCs and a related mechanism called HMSCs (High-level Message Sequence Charts) have been standardized [26] for specifying telecommunication software. A version of MSC called Sequence Diagram is a behavioral diagram type used in the Unified Modeling Language (UML) [10].

In all these settings, MSCs are used to capture system requirements. To move towards an implementation, one must obtain an executable specification which is related in some fashion to the MSC-based requirements. The key difficulty here, as identified in [14], is that the inter-object interactions described in form of MSCs must be related to -or synthesized as- executable specifications given in terms of intra-object behaviors, say, one state-chart for each object. This is a difficult problem and it has been studied in various limited contexts [1, 14, 17, 20].

In this paper, we propose using MSCs to construct executable specifications in a more direct fashion. The main idea is to use traditional methods to capture the control flow of the system components while using MSCs to describe the *non-atomic* component interactions. Among the various possibilities for describing the control flow in a multi-component system, we choose here the well-known model of synchronized product of transition systems; a network of labeled transition systems that synchronize on common actions. With suitable modifications one could easily use other related models as well.

We impose two restrictions on the control flow; a minor technical one that we will come to later but also a major one which requires that branchings in the control flow is effected by the components in a *local* fashion. In Petri net terms, this is the so called free choice property [9]. In particular, this restriction ensures that choices regarding which interactions to take part in are made by the components in a local fashion.

Starting with a network of labeled transition systems that synchronize on common actions, we refine each common abstract action γ involving a set of agents into a *transaction scheme* T_γ. Each such scheme is a guarded choice of MSCs. The life lines of the MSCs in T_γ will be from the set of agents participating in the common action γ. Each guarded MSC in T_γ, called a *transaction* will represent one possible interaction and will involve a complex flow of data and control signals. *When* a transaction scheme is to be executed is determined by the control flow in the high level product transition system. As to *which* transaction in T_γ will be chosen to be executed is determined by the guards which are propositional formulas built out of atomic propositions. The truth values of these atomic propositions, and hence those of the guards, will capture abstracted properties of the values of the variables associated with the agents. A central feature of the model is that both the control flow and the evaluation of the guards (which then leads to the execution of a specific transaction within a transaction scheme) are done in a distributed and asynchronous manner. In broad terms, this is our Communicating Transaction Processes (CTP) model.

Our model is in line with the emerging consensus that system-level design methods for embedded systems should be based on models of computation in which there is a clean separation of computational and communication features [11, 3, 12]. The CTP formalism basically uses finite state machines with data paths to model computational -and the attendant control flow- aspects while deploying guarded choices of MSCs to capture complex interactions between the different computational threads.

Our strategy of striking a balance between control flow and component interactions yields a model which is flexible, powerful and at the same time amenable to formal analysis and synthesis. Indeed, the problem of extracting an executable specification from the CTP model becomes very manageable and amenable to automation as we show in section 3. Our main point of reference for this work is the formalism of Live Sequence Charts [8] and more specifically the Play-in/Play-out approach [16] in which the component interactions are elaborated in a powerful way using the LSC language while the control flow information is completely suppressed. On the other hand, in models such as Petri nets and distributed transitions systems, the focus is on a detailed presentation of control flow while the only mechanisms for capturing component interactions are the atomic notions of synchronizing transitions and shared buffers.

An alternative way to use MSCs to capture system behavior is via HMSCs. However, an HMSC is just a presentation of a *collection* of MSCs. The problem of extracting an executable specification from an HMSC is a non-trivial one. There are a variety of choices available for the executable specification mechanism such as state charts [20], Petri nets [5] and networks of automata communicating through FIFOs [17, 1]. Many versions of this synthesis problem -i.e. deriving an intra-object executable specification from an HMSC- are not even decidable [17, 1, 5]. In contrast, as we shall show, we can extract an executable specification in the form of a finite Petri net from a CTP model effectively and in a manner that can be automated quite easily.

In the next section we introduce the CTP model while illustrating its main features with simple examples. In Section 3, we provide the operational semantics of the CTP model in terms of ordinary Petri nets. The key step in this process is converting the transaction schemes into an executable mechanism called event structures. In Section 4, we present a more detailed example based on the AMBA bus protocol in order to highlight the communicational aspects of the CTP model supported by the use of transaction schemes based on MSCs. In Section 5, we discuss behavioral properties and the means for determining these properties. In particular we present the notion of well-formed transaction schemes and illustrate its importance. In the subsequent section, we provide a more detailed comparison with the closely related formalism of LSCs. Section 7 reports our current efforts for building an experimental framework to enable the use of the CTP model to support the specification, verification and implementation of reactive embedded systems. The concluding section provides additional pointers to future research.

2 The CTP Model

Being based on MSCs, the CTP model captures non-atomic inter-process communications. However, in order to be amenable to efficient distributed implementation, this is combined with notations for describing intra-process control flow. As a starting example, consider the specification shown in Figure 1.[1] Each process repeatedly interacts with the other process and then performs some internal computational action. Note that the inter-process interaction and the internal actions have been separated into distinct units. A number of processes P will be involved in the execution of a chart. A process p which takes part in such an execution might next participate in a chart involving a different set of processes, say Q.

[1] We adopt the usual MSC convention that horizontal and downward sloping arrows denote message send-receives between two processes. Further, a $\square$ symbol on a single vertical line denotes an internal action (such as actions a and b in Figure 1). We denote a control state of a process as a circle.

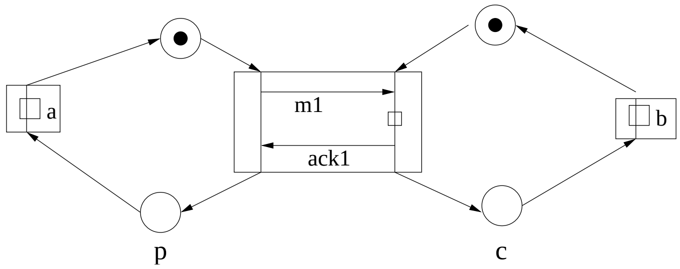

Fig. 1. Inter-process communication and intra-process control flow

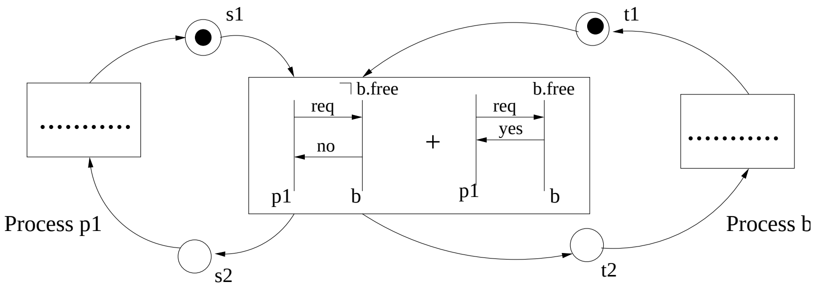

Fig. 2. Choice of Inter-process communication

The organization of interactions into the various units is up to the convenience of the designer. For example, in Figure 1 we could have made the actions a and b also to be part of the chart involving processes p and c. Note also that the example shown in Figure 1 is essentially a Petri net where the local control states in each process are the places of the net denoted by circles. Each top-level transition of this net is, in general, a collection of Message Sequence Charts at the refined level. A particular execution of the high-level transition, is an abstraction of the activity in which one of the charts associated with the high-level transition is chosen and executed. In the example of Figure 1 each net transition has a single chart associated with it. (An internal action is a degenerate chart involving just one process executing just one action). The choice as to which chart is executed -in case more than one chart is associated with a transition- is based on the value of the local variables of the processes. This is illustrated in Figure 2 where the choice is determined by the value of the variable $free$ belonging to process b). If $b.free$ holds once control reaches $s1$ and $t1$ respectively, we must execute the right-hand chart of Figure 2.

In general, the choice of which chart is executed at a particular net transition is a distributed one. Let the charts contained in a particular net transition be as shown in Figure 3. If $p1.data$ holds then chart 1 is ruled out. However, still we do not know whether chart 2 or chart 3 will be executed. This will depend on the value of variable $free$ in process b. As shown in Figure 3, each MSC associated with a net transition has a guard (which we will also refer to as a pre-condition). This guard is a distributed one in that it will in general involve propositions belonging to different processes participating in the MSC. At the end of the execution of a chart, the truth values of the various propositions will be set to new values in general.

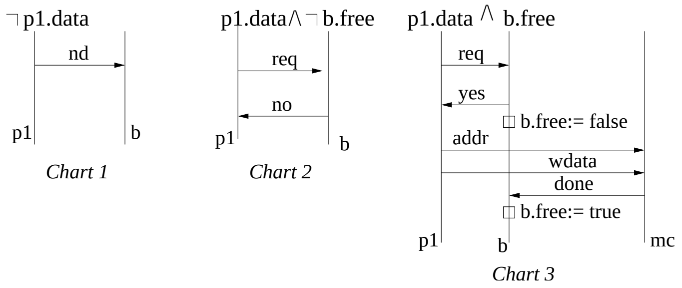

Fig. 3. Distributed nature of choice in a net transition

2.1 The Definition of the CTP Model

A product transition system is a network of sequential transition systems that synchronize on common actions. The CTP model is obtained by taking a restricted class of product transition systems and refining the common actions into collections of guarded MSCs called *transaction schemes*.

Fix a finite set of process names $\mathcal{P}$ with p, q ranging over $\mathcal{P}$. Fix also a finite set of labels Γ and a family $\{\Gamma_p\}_{p\in\mathcal{P}}$ with each Γ_p a subset of Γ and $\bigcup \Gamma_p = \Gamma$. This induces the function loc which assigns to each label in Γ the set of agents that participate in the execution of that action. This function is given by: $loc(\gamma) = \{p \mid \gamma \in \Gamma_p\}$. If $loc(\gamma) = \{p\}$ then γ will be called p-local action. The members of Γ will be treated as abstract action labels in the first step where we define the control flow model. In the second step they will be interpreted as transaction schemes and further elaborated. Γ_p is the set of (abstract) actions that the process p will participate in.

Anticipating the need to build guards in the second step, we also fix AP_p a finite set of atomic propositions, one for each p and set $AP = \bigcup_{p\in\mathcal{P}} AP_p$. If $P \subseteq \mathcal{P}$ then we let $AP_P = \bigcup_{p\in P} AP_p$. By convention, we shall write $AP_{\mathcal{P}}$ as AP. Each subset of AP_P will be called P-valuation. If $P = \{p\}$ is a singleton we will write p-valuation.

For each p let $TS_p = \langle S_p, \Gamma_p, \longrightarrow_p, init_p, V_{p,in}\rangle$ be a finite-state transition system over Γ_p with an initial p-valuation. In other words, S_p is a finite set of states, $init_p \in S_p$ is the initial state, $\longrightarrow_p \subseteq S_p \times \Gamma_p \times S_p$ denotes the transition relation and $V_{p,in} \subseteq AP_p$ is the initial valuation of atomic propositions in AP_p. In this paper we will be only interested in control flows in which the choices as to which transaction scheme that p will take part in is decided locally by p (free choice). Further, to avoid notational clutter, we will require that each member of Γ_p is the label of at most one transition in TS_p. These two restrictions on TS_p can be formalized as follows.

(1) if $s \xrightarrow{\gamma}_p s_1$, $s \xrightarrow{\gamma'}_p s_2$ and $s_1 \neq s_2$, then γ and γ' are p-local actions. Thus, $loc(\gamma) = loc(\gamma') = \{p\}$.
(2) If $s_1 \xrightarrow{\gamma}_p s_2$ and $s_3 \xrightarrow{\gamma}_p s_4$ then $s_1 = s_3$ and $s_2 = s_4$.

Definition 1 Product Transition System *A product transition system over* $(\{\Gamma_p, AP_p\})_{p\in\mathcal{P}}$ *is denoted as* $\{TS_p\}_{p\in\mathcal{P}}$ *where each*

$$TS_p = \langle S_p, \Gamma_p, \longrightarrow_p, init_p, V_{p,in}\rangle$$

is as specified above. As usual, the behavior of this product transition system is defined to be the global transition system $\langle S, \Longrightarrow, init, V_{in}\rangle$ *where:*

- $S = \prod_{p\in\mathcal{P}} S_p$
- $init = \prod_{p\in\mathcal{P}} init_p$
- $s \xRightarrow{\gamma} s'$ *iff* $s(p) \xrightarrow{\gamma}_p s'(p)$ *if* $p \in loc(\gamma)$ *and* $s(p) = s'(p)$ *otherwise. The notation* $s(p)$ *denotes local state of process* p *in global control state* s.
- $V_{in} = \bigcup_{p\in\mathcal{P}} V_{p,in}$

Next, we need to define transaction schemes. We begin with the standard notion of MSCs which we shall view, in the present context, as certain kinds of labeled partial orders. Their visual representation will be as shown in the various examples already. We shall use Σ_p to denote the set of actions executed by the process p. It consists of actions of the form $\langle p!q, m\rangle$, $\langle p?q, m\rangle$ and $\langle p, a\rangle$ where M is an alphabet of messages and Act is an alphabet of internal actions. The communication action $\langle p!q, m\rangle$ stands for p sending the message m to q and $\langle p?q, m\rangle$ stands for p receiving the message m from q. On the other hand, $\langle p, a\rangle$ is an internal action of p with a being the member of Act being executed. We set $\Sigma = \bigcup_{p\in\mathcal{P}} \Sigma_p$. We also denote the set of channels $Chan$ given by $Chan = \{(p, q) \mid p \neq q\}$.

Turning now to the definition of MSCs, we define Σ-labeled poset to be a structure $Ch = (E, \leq, \lambda)$ where $(E, \leq)$ is a poset and $\lambda : E \to \Sigma$ is a labeling function. For $X \subseteq E$ we define $\downarrow(X) = \{e' \mid e' \leq e \text{ for some } e \in X\}$. When $X = \{e\}$ is a singleton we shall write $\downarrow(e)$ instead of $\downarrow(\{e\})$. We say that X is *downclosed* in case $X = \downarrow(X)$. For $p \in \mathcal{P}$, we set $E_p = \{e \mid \lambda(e) \in \Sigma_p\}$. These are the events that p takes part in. Further, $E_{p!q} = \{e \mid e \in E_p \text{ and } \lambda(e) = \langle p!q, m\rangle \text{ for some } m \in M\}$. Similarly, $E_{p?q} = \{e \mid e \in E_p \text{ and } \lambda(e) = \langle p?q, m\rangle \text{for some } m \in M\}$. We define for any channel $c = (p, q)$, the communication relation R_c as: $(e, e') \in R_c$ iff $\mid \downarrow(e) \cap E_{p!q} \mid = \mid \downarrow(e') \cap E_{q?p} \mid$ and $\lambda(e) = \langle p!q, m\rangle$ and $\lambda(e') = \langle q?p, m\rangle$ for some message m.

An MSC (over $(\mathcal{P}, M, Act)$) is a Σ-labeled poset $Ch = (E, \leq, \lambda)$ which satisfies:

(1) $\leq_p$ is a linear order for each p where $\leq_p$ is $\leq$ restricted to $E_p \times E_p$.
(2) Suppose $\lambda(e) = \langle p?q, m\rangle$. Then $\mid \downarrow(e) \cap E_{p?q} \mid = \mid \downarrow(e) \cap E_{q!p} \mid$.
(3) For every p, q with $p \neq q$, $\mid E_{p?q} \mid = \mid E_{q!p} \mid$.
(4) $\leq = (\leq_{\mathcal{P}} \cup R_{Chan})^\star$ where $\leq_{\mathcal{P}} = \bigcup_{p\in\mathcal{P}} \leq_p$ and $R_{Chan} = \bigcup_{c\in Chan} R_c$.

This definition assumes a FIFO discipline for each channel. Other variations can also be dealt with easily. In what follows, we let $agents(Ch)$ denote the set of agents participating in the MSC $Ch = (E, \leq, \lambda)$ and define it as $agents(Ch) = \{p \mid E_p \neq \emptyset\}$.

Definition 2 Transaction Scheme *A Transaction Scheme γ is a finite collection of guarded Message Sequence Charts $\{[I^i : Ch^i]\}_{i=1}^k$. Each Ch^i is an MSC over $(\mathcal{P}, M, Act)$. Each I^i is of the form $\bigwedge_{p\in agents(Ch^i)} I_p^i$ where I_p^i is a propositional logic formula built from the propositions in AP_p.*

For each chart Ch^i in a transaction scheme, we have only mentioned a precondition. We have not specified the valuations of atomic propositions upon exiting from a chart. However, send and receive actions have a well-defined meaning. We can also assume that the internal actions are expressed in a standard imperative language. The operational semantics of this imperative language then lends a meaning to the internal actions. Consequently each event in a chart will have a well-defined effect on the truth-values of the local atomic propositions and as a sum total of these effects, we can associate with each chart an output valuation O^i. If more than one output valuation is possible, we can consider them as different transactions. Hence in what follows, we will assume that a transaction scheme is of the form $\{[I^i : Ch^i : O^i]\}_{i=1}^k$ over $(\mathcal{P}, M, Act)$.

Finally, we can now define a Communicating Transaction Processes (CTP) system model as follows.

Definition 3 CTP System Model *A CTP model is a product transition system $\{TS_p\}_{p\in\mathcal{P}}$ over $(\Gamma, \mathcal{P})$ where Γ is a finite set of transaction schemes over $(\mathcal{P}, M, Act)$. Further, for each $\gamma \in \Gamma$, $agents(\gamma) = loc(\gamma)$.*

Here $loc(\gamma)$ is as before where γ is viewed as an abstract action label in high level product transition system; $agents(\gamma)$ is the set of agents participating in some transaction associated with the transaction scheme γ. Let $\gamma = \{[I^i : Ch^i : O^i]\}_{i=1}^k$. Then $agents(\gamma) = \bigcup_{i=1,2,\ldots n} agents(Ch^i)$. Thus the restriction in the above says that the processes taking part in a high level transition in the control flow model are the same as the processes taking part in the transaction scheme associated with this high level transition. This restriction still allows the designer to reorganize the distribution of transactions across the various transaction schemes. Indeed, in the extreme case can one collapse the whole model into a single messy transaction scheme with just one control state for each process! A subclass of CTPs can be obtained by requiring $loc(\gamma) = agents(Ch^i)$ for each i above. In such CTPs one cannot arbitrarily rearrange the transactions.

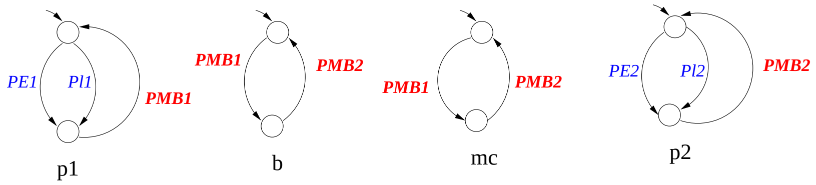

Fig. 4. CTP system model of Multiprocessor example (common actions are shown in bold)

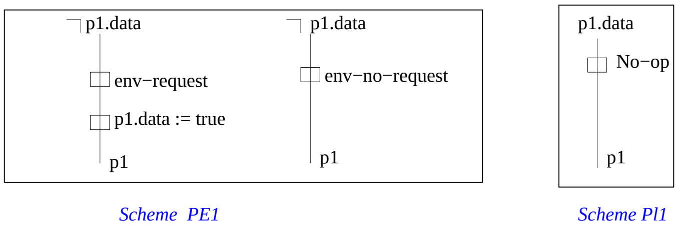

Fig. 5. Local Choices and Environment Interaction in Transaction Schemes of Fig. 4

2.2 A Simple Example

Consider two processors communicating with a shared memory via a bus. The bus controller serves as an arbiter for bus access and serializes the bus access requests by the two processors. The memory controller provides data to the processors for read requests and commits data for write requests. Two of the simple schemes of this system are shown in Figure 5 whereas the high-level control flow is as shown in Figure 4. The two processors are denoted by processes $p1$ and $p2$; b is the bus controller and mc is the memory controller.

The schemes *Pl1* and *PE1* are local schemes in which only $p1$ participates (refer Figure 5). They represent local choices. Scheme *Pl1* is executed when processor $p1$ has data to transfer ($p1.data$ is true). Thus, this scheme consists of a single degenerate MSC. The MSC consists of a single internal action which is a no-op. If processor $p1$ has no data to transfer, then scheme *PE1* is executed. This scheme consists of two MSCs. The choice of which chart is executed is made by the environment. If the environment (*i.e.* the application running on the processor) has data to transfer then $p1.data$ is set; otherwise it remains reset. In this simple example, whether *Pl1* or *PE1* is executed, the process $p1$ next participates in the same transaction scheme, namely, *PMB1*. In general however, this branching in the control flow could lead to different transaction schemes being chosen.

Since the processors have similar behavior, the scheme *PMB1* is identical to *PMB2* except that process $p1$ is replaced by $p2$. (Similar remarks hold for *PE1* and *PE2* as well as *Pl1* and *Pl2*) The scheme *PMB1* is the one shown earlier in

Figure 3. This scheme involves a decision by the bus controller b about granting bus access to $p1$. In Figure 3, $p1.data$ holds when $p1$ has data to transfer; $b.free$ holds when the bus is free for transfer. After the transfer the bus is set free. In this simple example, we have assumed that the bus is released after every access, and only write transfers are shown. We have also used our formalism to model more complex interactions such as burst transfers and split transfers, as discussed in Section 4.

3 The Petri Net Semantics

Our goal here is to provide an operational semantics for the CTP model in terms of Petri nets. A key step in our semantics is to combine the different guarded transactions within a transaction scheme into a single entity. This entity will consist of a parallel composition of computation trees; one computation tree for each process that participates in the transaction scheme. Finite labeled *event structures* [21] can be conveniently used for representing such a parallel composition. We define:

Definition 4 Event Structure *An* event structure *is a triple* $ES = (E, \leq, \#)$ *where* E *is a set of events,* $\leq \subseteq E \times E$ *is a partial ordering* causality *relation and* $\# \subseteq E \times E$ *is a* conflict *relation which is required to satisfy the following conditions: (a)* $\#$ *is irreflexive and symmetric, and (b) conflict is inherited via causality, that is* $(e_1 \# e_2 \wedge e_2 \leq e_3) \Rightarrow e_1 \# e_3$.

The idea is that in any execution if an event e occurs and $e' \leq e$ then e' must have occurred earlier in the same execution. On the other hand two events that are in conflict are mutually exclusive. They can never both occur in the same execution. Consequently, if e and e' are mutually exclusive and e'' causally depends on e' then e and e'' are mutually exclusive as well. This is captured by the fact that conflict is inherited via causality.

As a related notion, we define a Σ-labeled event structure to be a structure $ES = (E, \leq, \#, \Lambda)$ where $(E, \leq, \#)$ is an event structure and $\Lambda : E \to \Sigma$ is a labeling function. In diagrams, as illustrated in Figure 6, it will be convenient to represent the causality and conflict relation in a minimal fashion. To this end, we define the immediate causality relation $\lessdot$ and the immediate conflict relation $\#_\mu$ via:

- $e \lessdot e'$ iff $e < e'$ and for every e'', if $e \leq e'' \leq e'$ then $e = e''$ or $e'' = e'$.
- $e \#_\mu e'$ iff $(\downarrow(e) \times \downarrow(e')) \cap \# = \{(e, e')\}$.

Thus two events are in immediate conflict if they are in conflict and their being in conflict can not be attributed to an earlier conflict that is inherited via the causality relation.

The event structure corresponding to the transaction scheme in Figure 2 is shown in Figure 6. The minimal causal relationship is captured by unidirectional arrows. The minimal conflict relation $\#_\mu$ is captured by curved bidirectional arrows. The way the minimal and maximal events of this event structure are connected to the input and output control states of the transaction scheme are also shown.

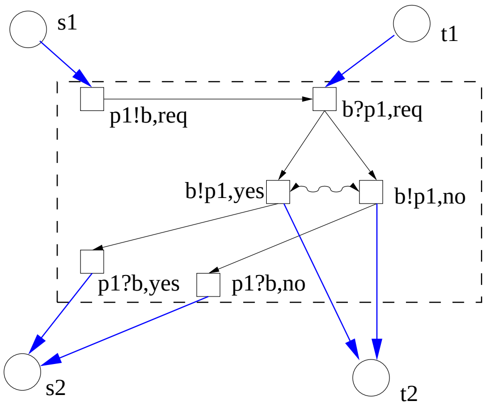

Fig. 6. Event Structure for Transaction scheme in Figure 2 (shown in dashed box)

A (labeled) event structure is accompanied by a natural dynamics. A state is a set of events that have occurred so far along an execution. States are usually referred as configurations. Formally, a configuration c of the event structure $ES = (E, \leq, \#)$ is subset of E which is downclosed and conflict-free. In other words $\downarrow(c) = c$ and $(c \times c) \cap \# = \emptyset$. The empty set is a configuration; it is the initial configuration.

Let $\mathcal{C}_{ES}$ be the set of (finite) configurations of ES. The event e is *enabled* at the configuration c if e is not in c and $c \cup \{e\}$ is also a configuration. This leads to the transition relation $\longrightarrow_{ES} \subseteq \mathcal{C}_{ES} \times E \times \mathcal{C}_{ES}$ where $c \xrightarrow{e}_{ES} c'$ iff e is enabled at c and $c' = c \cup \{e\}$. Thus we can associate the transition system $TS_{ES} = (\mathcal{C}_{ES}, \longrightarrow_{ES}, \emptyset)$ with the event structure ES. These ideas extend in the expected manner to labeled event structures.

3.1 Constructing Event structures

In order to define our operational semantics, we first recall that $AP = \bigcup_{p \in \mathcal{P}} AP_p$ is the set of atomic propositions. Let γ be a *transaction scheme* (refer Definition 2) of the form $\gamma = \{I^i : Ch^i : O^i\}_{i=1}^{n}$ where each I^i is a propositional formula built out of AP, each $Ch^i = (E^i, \leq^i, \lambda^i)$ is a chart over $(\mathcal{P}, M, Act)$ and each O^i is a subset of AP. We let $\gamma^i = [I^i : Ch^i : O^i]$ for each i and call I^i, the *input guard*, Ch^i the *body* and O^i the *output valuation* of the transaction T^i. We will assume without loss of generality that the sets $\{E^i\}_{i=1,\ldots,n}$ are pairwise disjoint.

We construct the labeled event structure $ES_\gamma = (E, \leq \#, \lambda)$ to be associated with a transaction scheme γ as follows. The set of events E is obtained from the event sets E^i ($i = 1, \ldots, n$) but after identifying events that have isomorphic pasts. Consequently we start with a set X whose elements will be of the form (e, i, P, V_P) where $e \in E^i$, $P = \{p \mid \exists e' \in E^i_p \,\text{and}\, e' \leq^i e\}$ and V_P is a P-valuation such that $V_P \models \bigwedge_{p \in P} I^i_p$. Note that E^i_p is the set of events in E^i in which p participates, that is, for any event $e \in E^i_p$, the label $\lambda^i(e)$ is of the form $\langle p!q, m\rangle$ or $\langle p?q, m\rangle$ or $\langle p, a\rangle$.

Actually, the second and third components in (e, i, P, V_P) are redundant but we will carry them for convenience. Next let $x = (e, i, P, V_P)$ and $y = (d, j, Q, V_Q)$ be in X. Then $x \equiv y$ iff $\downarrow(e)$ in Ch^i is isomorphic to $\downarrow(d)$ in Ch^j in the obvious sense. We shall denote the $\equiv$-equivalence class containing x as $[x]$.

- *Set of Events:* We define E, the set of events of ES_γ to be the $\equiv$-equivalence classes of X. Thus, $E = \{[x] \mid x \in X\}$. Thus for the scheme shown in figure 2, the two events of $p1$ that send a request message req are equivalent as also the two corresponding receive events.
- *Causality Relation:* Let $[x], [y]$ be in E. Then $[x] \leq [y]$ iff there exists (e, i, P, V_P) in $[x]$ and (d, j, Q, V_Q) in $[y]$ such that $i = j$, $e \leq^i d$ and $V_Q \cap AP_P = V_P \cap AP_P$.
- *Conflict Relation:* First we define the relation $\widehat{\#}$ to be the least subset of $E \times E$ which satisfies the following. Suppose $[x], [y] \in E$ are such that $[x] \not\leq [y]$ and $[y] \not\leq [x]$. Furthermore, there exist (e, i, P, V_P) in $[x]$ and (d, j, Q, V_Q) in $[y]$ such that $e \in E^i_p$ and $d \in E^j_p$ for some p but $i \neq j$. Then $[x]\widehat{\#}[y]$. We now define the conflict relation $\#$ as the least subset of $E \times E$ which (a) contains $\widehat{\#}$, (b) is a symmetric relation, and (c) inherits through causality, that is, $[x]\#[y]$ and $[y] \leq [z]$ implies $[x]\#[z]$.
- *Labeling Function:* Finally, the labeling function Λ is given by:

$$\Lambda([(e, i, P, V_P)]) = \lambda^i(e)$$

Lemma 1 *$ES_\gamma = (E, \leq, \#, \Lambda)$ is a labeled event structure.*

Proof: Due to the isomorphism condition imposed in the definition of $\equiv$, it is easy to observe that $\leq$ is a partial ordering relation. From the definition of the relation $\#$ it is symmetric and is inherited via $\leq$. We need to show that it is irreflexive. Assume for contradiction that there exists $[x]$ such that $[x]\#[x]$. In this case it is not difficult to see there exist $[y]$ and $[z]$ such that $[y]\widehat{\#}[z]$ and $[y] \leq [x]$ and $[z] \leq [x]$. This implies there exist (e, i, P, V_P) in $[y]$ and $(e1, i, P1, V_{P1})$ in $[x]$ such that $e \leq^i e1$. Further, there exist (d, j, Q, V_Q) in $[z]$ and $(d1, j, Q1, V_{Q1})$ in $[x]$ such that $d \leq^j d1$. Then by the definition of the $\equiv$ relation, it follows that there exists $(d', i, Q', V_{Q'})$ in $[z]$ such that $d' \leq^i e1$. But then from the definition of $\widehat{\#}$ it follows that there exists p such that $e \in E^i_p$ and $d' \in E^i_p$. This leads to $e \leq^i d'$ or $d' \leq^i e$ which in turn leads to $[y] \leq [z]$ or $[z] \leq [y]$ contradicting $[y]\widehat{\#}[z]$. The fact that the labeling function is well-defined is obvious. $\square$

3.2 The Petri Net Semantics

We construct the Petri net semantics for the CTP model in three steps. First we convert each labeled event structure yielded by a transaction scheme into an acyclic net (without an initial marking). We then merge these nets with the high level control flow net. As a last step we refine local control states and the transitions to expose information about the valuations of the atomic propositions.

From Event Structure to Acyclic Net First, let γ be a transaction scheme and $ES_\gamma = (E, \leq, \#, \lambda)$ be its event structure representation. For $e \in E$ we set $proc(e) = p$ if there exists (x, i, P, V_P) in e such that $x \in E_p^i$. It is easy to see that $proc$ is a well-defined function, and it is also easy to check that if $e \widehat{\#} e'$, $proc(e) = proc(e')$.

Now, we define the net associated with our event structure. Before doing so, note that the *minimal* causality relation of event structure ES_γ is denoted as $\lessdot$; the minimal conflict relation is denoted as $\#_\mu$. From the construction of ES_γ it follows easily that that if $e \#_\mu e'$ then $proc(e) = proc(e')$. Furthermore, $\#_\mu$ is transitive (and symmetric). Hence in what follows, while writing $\#$ instead of $\#_\mu$ for convenience, we will let $[e]_\#$ denote the set of events given by $[e]_\# = \{e\} \cup \{e' \mid e \#_\mu e'\}$.

We define the net representation $ES_\gamma = (E, \leq, \#, \lambda)$ as the acyclic net $N_\gamma = (B_\gamma, E_\gamma, F_\gamma)$ where:

(1) The set of transitions $E_\gamma = E$.
(2) The set of places B_γ and the flow relation F_γ are the least sets which satisfy:
   (i) Suppose $e \lessdot e'$ and $proc(e) \neq proc(e')$. Then $(e, e') \in B_\gamma$, $(e, (e, e')) \in F_\gamma$, and $((e, e'), e') \in F_\gamma$.
   (ii) Let $e \lessdot e'$ and $proc(e) = proc(e')$. Then $(e, [e']_\#) \in B_\gamma$ and $(e, (e, [e']_\#)) \in F_\gamma$ and $((e, [e']_\#), e'') \in F_\gamma$ for every e'' in $[e']_\#$.

The net representation of the event structure of Figure 6 is shown in Figure 7.

Merging the Control Flow Let $\text{TP} = \{TS_p\}_{p \in \mathcal{P}}$ be a CTP over $(\Gamma, \mathcal{P})$ where Γ is a finite set of transaction schemes over $(\mathcal{P}, M, Act)$. Let $TS_p = (S_p, \Gamma_p, \longrightarrow_p, init_p, V_{p,in})$ be the transition system associated with transaction process p (note that $V_{p,in}$ is the initial p-valuation). For each transaction scheme γ in Γ let ES_γ be its event structure representation and $N_\gamma = (B_\gamma, E_\gamma, F_\gamma)$, the net associated with ES_γ.

For convenience we will denote the set of pre and post control states of the transaction scheme γ as ${}^\bullet\gamma$ and $\gamma^\bullet$ respectively and define these sets as:
${}^\bullet\gamma = \{s \mid \gamma \in \Gamma_p \text{ and } s \xrightarrow{\gamma}_p s' \text{ for some } s, s' \in S_p\}$.
$\gamma^\bullet = \{s' \mid \gamma \in \Gamma_p \text{ and } s \xrightarrow{\gamma}_p s' \text{ for some } s, s' \in S_p\}$.

We can now carry out the second step in providing the operational semantics.

The *control flow Petri net* of TP is the Petri net

$$\mathrm{CFN}_{\mathrm{TP}} = (S_{\mathrm{TP}}, T_{\mathrm{TP}}, F_{\mathrm{TP}}, M_{in,\mathrm{TP}})$$

where:

- $S_{\mathrm{TP}} = \bigcup\{S_p \mid p \in \mathcal{P}\} \cup \bigcup\{B_\gamma \mid \gamma \in \Gamma\}$.
- $T_{\mathrm{TP}} = \bigcup\{E_\gamma \mid \gamma \in \Gamma\}$
- $F_{\mathrm{TP}} = \bigcup_{\gamma \in \Gamma}(F_\gamma) \cup$
  $\{(s,e) \mid e \in min(E_\gamma), s \in {}^\bullet\gamma \cap S_p, proc(e) = p\} \cup$
  $\{(e,s') \mid e \in max(E_\gamma), s' \in \gamma^\bullet \cap S_p, proc(e) = p\}$)
- $M_{in,\mathrm{TP}}(z) = 1$ if there exists p s.t. $z = init_p$ (the initial state of some p). Otherwise $M_{in,\mathrm{TP}}(z) = 0$.

By $min(E_\gamma)$ ($max(E_\gamma)$) we mean the set of minimal (maximal) elements under the causality relation of the event structure ES_γ.

The control flow net of a CTP description captures the behaviors in the individual processes with one major caveat. For events which are in minimal conflict, it does not expose the valuations of the atomic propositions which resolve the conflict. As an example, consider the event structure of Figure 6 and its net representation shown in Figure 7. Now, consider the place marked in bold in Figure 7; this place has two outgoing flow arcs leading to two events in minimal conflict. Here, the control flow net contains infeasible behaviors not allowed by the transaction scheme of Figure 2. This is because the control flow net of Figure 7 does not capture the condition which needs to be evaluated to decide which of the two conflicting events is executed (in this case, the condition is $b.free$). The simplest solution is to annotate the flow arcs with this condition (*i.e.*, the two arcs should be annotated with $b.free$ and $\neg b.free$).[2] However adding such annotations does not give an executable model of the allowed behaviors for a CTP. To construct such an executable model, we need to systematically expose the data dependencies, that is, the valuation of atomic propositions in the places and transitions of the control flow net. This is now done by constructing a Petri net corresponding to any CTP specification.

Constructing the Petri Net Let $\mathrm{CFN} = (S_{\mathrm{TP}}, T_{\mathrm{TP}}, F_{\mathrm{TP}}, M_{in,\mathrm{TP}})$ be the control flow net of TP, a CTP. Then $\mathrm{PN}_{\mathrm{TP}}$ is the Petri net representation of TP and it is the Petri net $\mathrm{PN}_{\mathrm{TP}} = (S, T, F, M_{in})$ where S, T and F are the least set of elements satisfying the following conditions:

- Suppose s is in S_p. Then (s, V_p) is in S where V_p is a p-valuation. Next let γ be in Γ and $N_\gamma = (B_\gamma, E_\gamma, F_\gamma)$ be the net associated with ES_γ and $(x,y) \in B_\gamma$. Now suppose $(e, i, P, V_P) \in x$. Then $((x,y), V_P)$ is in S.
- Let γ be in Γ and $N_\gamma = (B_\gamma, E_\gamma, F_\gamma)$ be the net associated with ES_γ and $x \in E_\gamma$. Suppose $(e, i, P, V_P) \in x$. Then (x, V_P) is in T.

[2] One could capture this easily using Colored Petri nets [19] but this would entail an additional intermediate description.

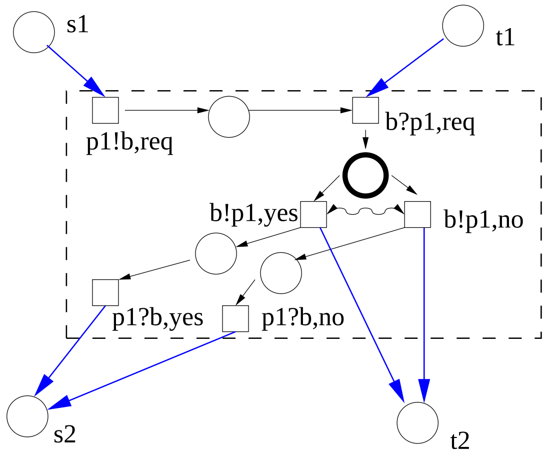

Fig. 7. Acyclic Net for Transaction scheme in Figure 2 (shown in dashed box)

– Suppose $(s, x) \in F_{\mathrm{TP}}$ with $s \in S_p$ for some p and $(e, i, \{p\}, V_p) \in x$. Then $((s, V_p), (x, V_p))$ is in F. Also, suppose $(x, s') \in F_{\mathrm{TP}}$ with $s' \in S_p$ for some p and $(e, j, Q, V_Q) \in x$ and O^j is the output valuation of the transaction $[I^j : Ch^j : O^j]$. Then $((x, V_Q), (s', V_p))$ is in F where $V_p = O^j \cap AP_p$. Finally, let $((x, y), V_P) \in S$. Then $((x, V_P), ((x, y), V_P))$ is in F. Furthermore, $(((x, y), V_P), (y, V_Q)) \in F$ provided $proc(x) \neq proc(y)$ and (y, V_Q) is in T and $V_Q \cap AP_P = V_P$. In case $proc(x) = proc(y)$ then $(((x, y), V_P), (z, V_Q)) \in F$ provided $z \in y$ and $V_Q \cap AP_P = V_P$. In general, y denotes a set of events in minimal conflict and belonging to the same process.
– $M_{in}(z) = 1$ if $z = (init_p, V_{p,in})$ for some p. Otherwise $M_{in}(z) = 0$.

For example, the Petri net fragment for the transaction scheme of Figure 2, together with the refined representation of its surrounding control places will be as shown in Figure 8. Here for convenience we have assumed that the output guard for both the transactions is $b.free$.

This concludes the construction of the Petri net to be associated with a CTP. The execution semantics of a CTP is then just the usual execution semantics of its associated Petri net.

4 Specifying the AMBA bus protocol

In this section, we present a non-trivial example to show the use of the CTP as a modeling 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

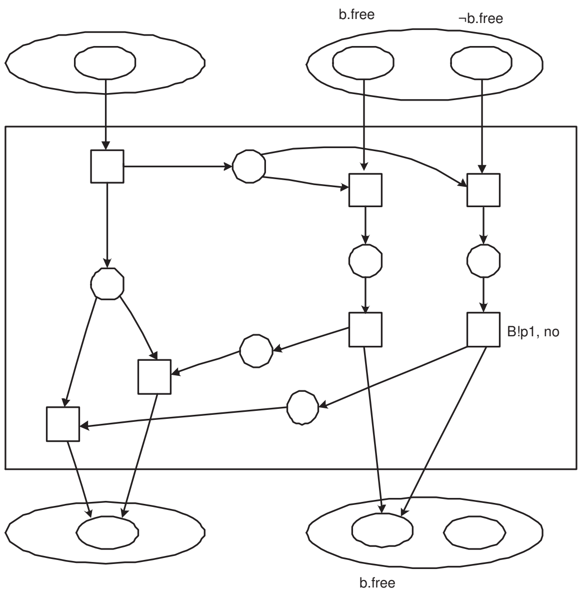

Fig. 8. Petri net fragment for Transaction scheme in Figure 2

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 high level transition systems of the individual processes are shown in Figure 9. In the diagram, the constituent guarded transactions of the various transaction schemes have not been shown.

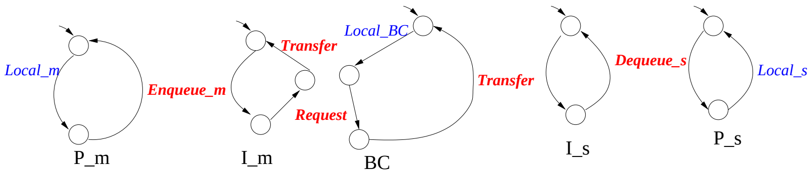

Fig. 9. CTP model of interfaces between two embedded co-processors. Common actions are shown in bold. Wherever possible, labels of repeated occurrences of a common action have been shared to reduce visual clutter.

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 [2]. 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 [7]. These interfaces were originally designed for data transfer between co-processors connected to a common bus running the PI-Bus protocol. Here instead we shall be using the AMBA bus protocol.

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. The other three transaction schemes denote the following interactions. $Enqueue_m$ 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 $Transfer$ where

$$agents(Transfer) = \{I_m, I_s, BC\}$$

In particular, this will show our formal specification of the AMBA bus protocol. 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 10. 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. 10. 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).

Message Sequence Charts The transaction scheme $Transfer$ is collection of MSCs, one for each of the following mutually exclusive conditions.[3]

(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 11), 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 12. This corresponds to the scenario where wait cycles are initiated (note that $waitcnt = 0$) for some transfer, since the queue at I_s is full. Note that the first three actions by I_m in this chart are the same as Figure 11. 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.

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 13.

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 14.

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 15.

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

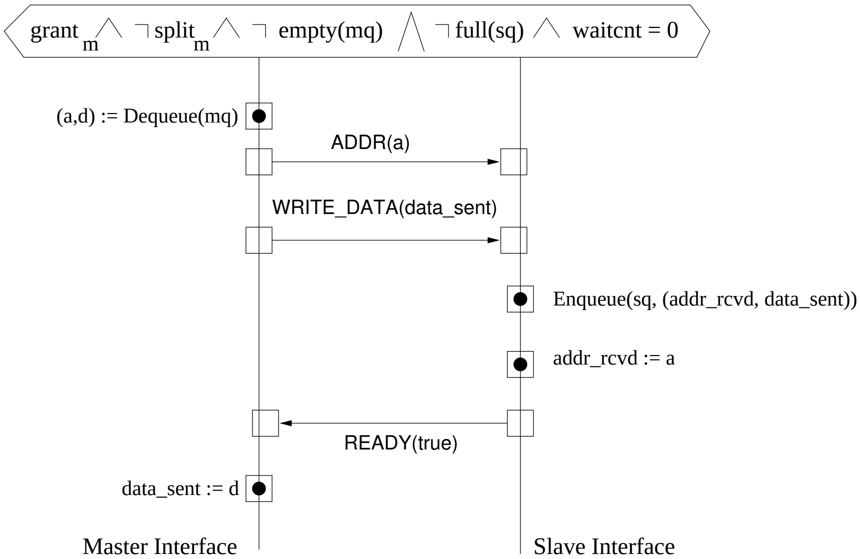

Fig. 11. Normal data transfer between master and slave interface

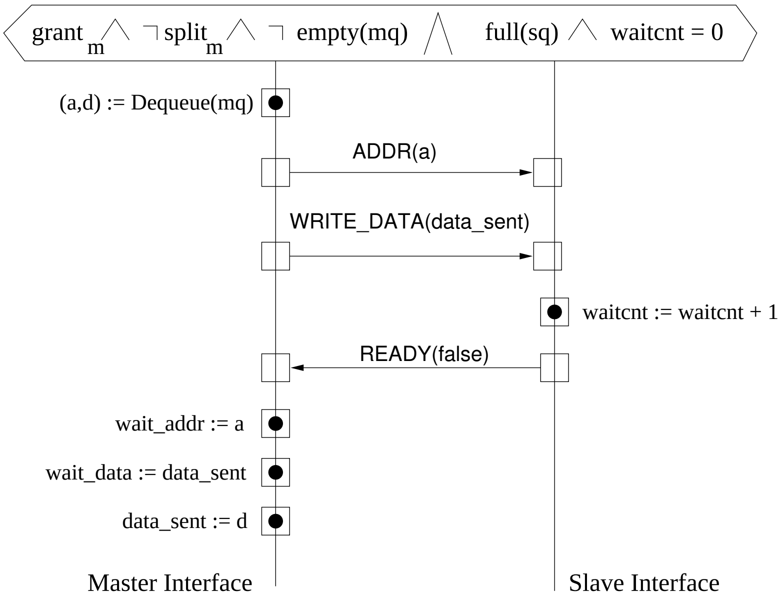

Fig. 12. Initiation of wait cycles

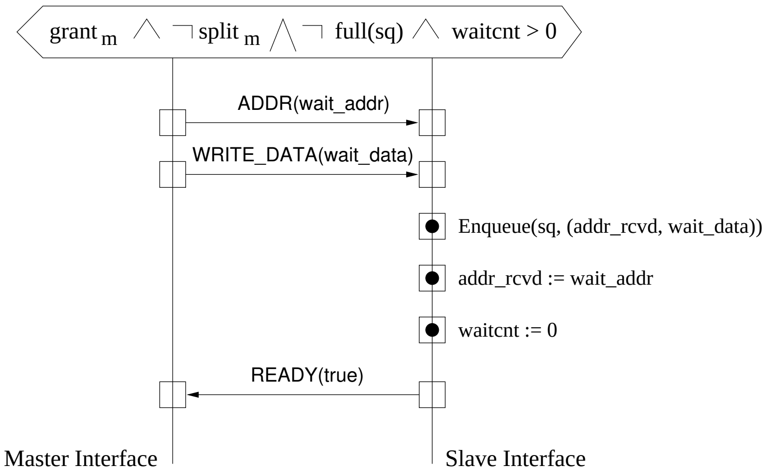

Fig. 13. The last wait cycle

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. 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 15, and the resetting of *waitcnt* to zero.

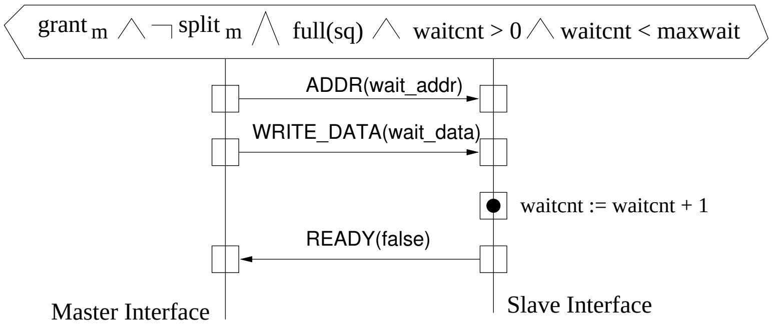

Fig. 14. A wait cycle which is not the last

Remark 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

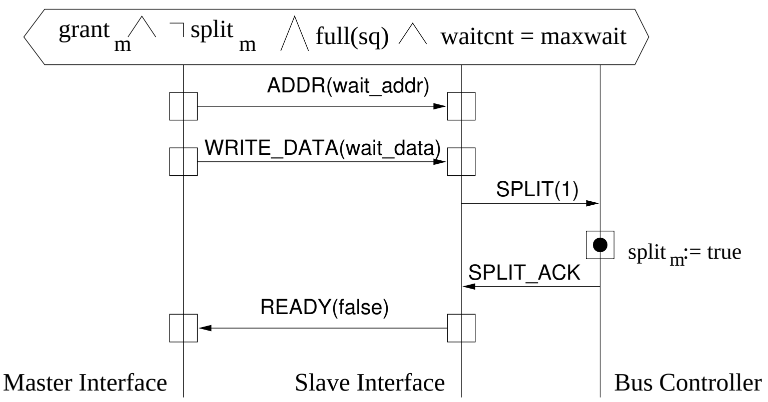

Fig. 15. Initiation of split transfer

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. An elegant way of modeling masters and slaves is to treat all masters as one process class, and all slaves as another process class. The individual masters and slaves then correspond to concrete objects of these classes. This requires extending our model to handle objects, classes and subclasses. We are currently pursuing research in this direction.

5 Behavioral Properties of CTPs

In this section, we introduce some important behavioral properties of the CTP model and the techniques currently available for determining these properties.

5.1 Well-formed Transaction Schemes

For pragmatic reasons, our definition of the CTP model imposes almost no syntactic restrictions. As a result, one can easily specify behaviors which are problematic from both specification and implementation standpoints. For instance, consider the transaction scheme shown in Figure 16 and its associated event structure. If the control flow enables this transaction scheme with the valuation $\{\neg A, B\}$, there will be a deadlock and no event in the associated event structure will execute. On the other hand if the valuation is $\{A, \neg B\}$ then the send events $\langle p!q, m1\rangle$ and $\langle q!p, m2\rangle$ can execute with no order after which there will be a deadlock. Thus local deadlocks can arise due to incomplete specification of the transaction schemes. As a method for detecting and eliminating such undesirable behaviors, we propose the notion of *well-formed* transaction schemes. Intuitively, this notion says that in the locality of a transaction scheme, the maximal executions of the event structure associated with the transaction scheme are precisely the executions of the transactions mentioned in the transaction scheme.

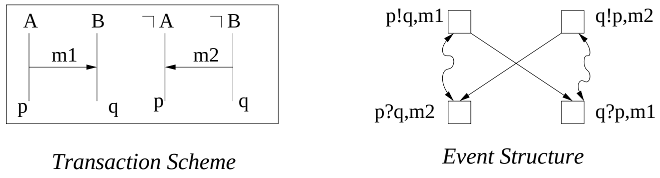

Fig. 16. A Transaction Scheme which is not well-formed

Definition 5 Well-formed Transaction Scheme *Let* $\mathrm{T} = \{T^i = [I^i : Ch^i : O^i]\}_{i=1,2,\ldots,n}$ *be a transaction scheme and* $ES_\mathrm{T} = (E, \leq, \#, \Lambda)$ *be its event structure representation. For a configuration (a downclosed conflict-free subset of events) c of* ES_T*, we let* $ES_{c,\mathrm{T}}$ *be the sub-event structure induced by c; it is the event structure* $(c, \leq_c, \#_c)$ *where* $\leq_c(\#_c)$ *is* $\leq(\#)$ *restricted to c. Let* $MAXC_\mathrm{T}$ *be the sub-event structures of* ES_T *induced by the set of maximal configurations of* T*. Then, transaction scheme* T *is said to be* well-formed *iff there exists a bijection* $f : \{1, 2, \ldots, n\} \rightarrow MAXC_\mathrm{T}$ *such that* Ch^i *is isomorphic to* $f(i)$ *for each* i *in* $\{1, 2, \ldots, n\}$*.*

Each transaction scheme can be effectively analyzed to determine if it is well formed. Clearly the transaction scheme shown in Figure 16 is not well-formed. We are *not* advocating the notion of well-formedness as mandatory but we believe it is a useful criterion using which certain types of incomplete and inconsistent specifications at the level of transaction schemes can be caught. It should also be clear that well-formedness alone will not suffice to guarantee sound implementations. For instance if the behaviors of a transaction scheme exhibit intra-process non-determinism then hardware implementation can be problematic.

5.2 Behavioral Properties

Let TP be a CTP and $\mathrm{N_{TP}}$ be its Petri net representation. We shall assume the standard behavioral notions for Petri nets here [9]. We will say that that TP is *transaction-deterministic* if for every reachable marking M of $\mathrm{PN_{TP}}$, if x and y are events belonging to the event structure associated with a transaction scheme in TP and both x and y are enabled at M then $proc(x) \neq proc(y)$. Consequently x and y can occur causally independent of each other at M. Thus transaction-determinism guarantees during the course of executing events taken from a transaction scheme, there will be no conflict. We will also say that TP is *bounded* in case its Petri net is bounded (has only a finite set of reachable markings).

We say that a transaction scheme is *anchored* in case each of its transactions is anchored. A transaction is anchored if its associated MSC, say, $Ch = (E, \leq, \lambda)$ has a least element e_{in} and greatest element e_{fin} and further more,there exists p such that $e_{in}, e_{fin} \in E_p$. Thus the transaction is initiated and terminated by a single agent. An interesting observation here is that if each transaction scheme in a CTP is anchored, then the CTP is bounded.

Via the Petri net semantics, the notions of TP being *live* and *dead-lock free* can also be defined. Clearly, all these properties are decidable since the corresponding problems for Petri nets are decidable. We are currently studying how efficient decision procedures can be developed by exploiting the additional structure provided by the CTP model.

6 Connection to Live Sequence Charts (LSC)

Our CTP formalism serves as a high level executable specification language based on Message Sequence Charts. Recently, Damm and Harel have developed the Live Sequence Charts (LSC) formalism which is also a MSC based modeling language. A powerful execution framework for LSCs based on the so called Play-in/Play-out approach is also being developed by Harel and his collaborators [13, 15]. In this section, we explore the connections between the CTP and LSC formalisms, namely (a) how to interpret LSCs over the CTP model and (b) how to translate CTP models to the LSC language.

The basic feature of LSCs is that it has two types of charts, namely *existential* and *universal* charts. The universal charts are used to specify requirements that *all* the possible system runs must satisfy. A universal chart typically contains a *pre-chart* followed by a main chart to capture the requirement that if along any run, the scenario depicted in the pre-chart occurs then the system *must* also execute the main chart. Existential charts specify sample interactions, typically between the system components and the environment that at least one system run must satisfy. Existential charts can be used to specify system tests and illustrate typical unrestricted runs. The LSC formalism also uses *cold* and *hot* conditions which are in some sense provisional and mandatory guards. If a cold condition holds during an execution then control is intended to pass to the location immediately after the cold condition. If it is false then the chart-context in which this condition occurs is exited. A hot condition, on the other hand, must always be true. If an execution reaches a hot condition which evaluates to false then this signals the violation of requirement and the system is supposed to abort. Thus we can attach the constant *false* condition at the end of chart Ch to capture the requirement that Ch must never occur. In other words, Ch is a forbidden scenario. On the other hand, cold conditions can be used to program if-the-else constructs. Similarly we can also specify events to be hot or cold.

It will be convenient to break down the features of the LSC language into simple units and present them individually. Assuming the notations and terminology developed in the previous section, we define a basic *universal* LSC (over $(\mathcal{P}, M, Act)$) with a *pre-chart* as a structure $[PCh, BCh]$ where:

(1) $PCh = (E_{PCh}, \leq_{PCh}, \lambda_{PCh})$ is a MSC called the *pre-chart*.
(2) $BCh = (E, \leq, \lambda)$ is a MSC called the *body* with $E_{PCh} \cap E = \emptyset$.
(3) $[PCh, BCh]$ denotes $PCh \circ BCh$, the asynchronous concatenation of PCh with BCh. This is in keeping with the asynchronous nature of our CTP model; [8] mentions a variant involving synchronous concatenation. Thus, strictly speaking, we consider an asynchronous version of the LSC formalism [8].
(4) $agents(min(BCh)) \subseteq agents(PCh)$.

In order to explain the last condition given above, recall that $min(BCh)$ is the set of minimal events in BCh. The last condition is intended to ensure that in the asynchronous concatenation of PCh followed by BCh, every event of BCh will have a causal predecessor in PCh. As might be expected, we define the asynchronous concatenation $Ch^1 \circ Ch^2$ of two MSCs $Ch^1 = (E^1, \leq^1, \lambda^1)$ and $Ch^2 = (E^2, \leq^2, \lambda^2)$ with $E^1 \cap E^2 = \emptyset$ as the MSC $Ch = (E, \leq, \lambda)$ where $E = E^1 \cup E^2$ and $\lambda(e) = \lambda^1(e)$ $(\lambda^2(e))$ if e is in E^1 (E^2). Finally $\leq$ is the least partial ordering relation over E which contains $\leq^1$ and $\leq^2$ and satisfies: if $e \in E^1_p$ and $e' \in E^2_p$ for some p then $e \leq e'$.

We define a basic *universal* chart with *pre-condition* as a structure $[P, \varphi, BCh]$ where φ is a propositional formula built out of AP_P called the pre-condition and BCh is a chart called the body such that $agents(min(BCh)) \subseteq P$. Basic *existential* charts denoted $\langle PCh, BCh\rangle$ with a pre-chart as well as basic existential charts with pre-conditions denoted $\langle P, \varphi, BCh\rangle$ can be defined in a similar fashion. Neither existential nor universal charts with post-charts are interesting. We however define a basic universal (existential) chart with a post-condition as the structure $[BCh, P, \varphi]$ $(\langle BCh, P, \varphi\rangle)$ where, as before, φ is a propositional formula built out of AP_P and $agents(max(BCh)) \subseteq P$. Intuitively such charts denote the property that if BCh is executed, φ must (may) hold.

A cold condition is a basic existential chart with a pre-condition whose body chart is empty. A hot condition is a basic universal chart with a post-condition whose body chart is empty. LSCs can now be inductively obtained by starting with the basic charts and allowing the body itself to be an LSC. Viewing the resulting class of LSCs as atomic assertions, one can obtain LSC specifications by forming boolean combinations of these atomic assertions.

6.1 When does a CTP model satisfy a LSC specification ?

We now interpret the basic LSC specifications over CTPs. It is easy to extend this interpretation to more complicated LSC specifications.

Let $\mathrm{TP} = \{TS_p\}_{p\in\mathcal{P}}$ be a CTP over $(\Gamma, \mathcal{P})$ where Γ is a finite set of transaction schemes over $(\mathcal{P}, M, Act)$. Let $\mathrm{PN}_{\mathrm{TP}}$ be the Petri net representation of

TP, constructed from Σ-labeled event structures representing the transaction schemes. Let $\Longrightarrow$ be the labeled transition relation defined over the reachable markings of $\mathrm{PN_{TP}}$ given by: $M \overset{\alpha}{\Longrightarrow} M'$ iff there exists a transition t of $\mathrm{PN_{TP}}$ such that t is enabled at M and M' is the resulting marking when t occurs at M. Furthermore, $\Lambda(t) = \alpha$ where Λ is the obvious labeling function that assigns to each transition of $\mathrm{PN_{TP}}$, a label in Σ (the set of labels of events appearing in the transaction schemes). This transition relation $\Longrightarrow$ is extended to Σ-sequences in the obvious way and this extension will be also be denoted as $\Longrightarrow$. Next let $Ch = (E, \leq, \lambda)$ be an MSC. Then λ applied pointwise to a linearization of $(E, \leq)$ yields a member of Σ^*. We let $lin(Ch)$ be the subset of Σ^* obtained this way, and refer to it also, by abuse of terminology, as the linearizations of Ch. We define Σ_{Ch} to be the subset of Σ given by $\Sigma_{Ch} = \{\lambda(e) \mid e \in E\}$. Finally, if $\sigma \in \Sigma^*$ and $\Sigma' \subseteq \Sigma$ then $\lceil_{\Sigma'} (\sigma)$ is the Σ' projection of σ. For an MSC Ch we will often write $\lceil_{Ch}$ instead of $\lceil_{\Sigma_{Ch}}$. We are now prepared to interpret the basic LSC specifications over CTPs.

(1) Let $[PCh, BCh]$ be a basic universal chart with a pre-chart. Then TP satisfies $[PCh, BCh]$ iff *every* reachable marking (s_0, V_0) of $\mathrm{PN_{TP}}$ satisfies the following condition. Suppose $M_0 \overset{\sigma_0}{\Longrightarrow} M_1$ such that $\lceil_{PCh} (\sigma_0)$ is in $lin(PCh)$. Then for *every* $M_1 \overset{\sigma_1}{\Longrightarrow} M_2$, there exists $M_2 \overset{\sigma_2}{\Longrightarrow} M_3$ such that a prefix of $\lceil_{\Sigma'} (\sigma_0\sigma_1\sigma_2)$ corresponds to a member of $lin(PCh \circ BCh)$ where $\Sigma' = \Sigma_{PCh} \cup \Sigma_{BCh}$. Thus, this universal requirement demands that whenever (a linearization of) PCh has been executed then this *must* be followed by an execution of (a linearization of) BCh.
(2) Next suppose $[P, \varphi, BCh]$ is a basic universal chart with a pre-condition. Then TP satisfies $[P, \varphi, BCh]$ iff *every* reachable marking $M_0 = (s_0, V_0)$ of $\mathrm{PN_{TP}}$ satisfies the following condition. Suppose $V_0 \models \varphi$. Then for every $M_0 \overset{\sigma_0}{\Longrightarrow} M_1$ there exists $M_1 \overset{\sigma_1}{\Longrightarrow} M_2$ such that a prefix of $\lceil_{BCh} (\sigma_0\sigma_1)$ is a member of $lin(BCh)$. Hence this universal requirement demands that whenever φ holds then this *must* be followed by an execution of BCh.
(3) Next suppose $\langle PCh, BCh\rangle$ is a basic existential chart with pre-chart. Then TP satisfies $\langle PCh, BCh\rangle$ iff there *exists* a reachable marking M_0 and $M_0 \overset{\sigma_0}{\Longrightarrow} M_1$ such that a prefix of $\lceil_{\Sigma'} (\sigma_0)$ contains a member of $lin(PCh \circ BCh)$. As before $\Sigma' = \Sigma_{PCh} \cup \Sigma_{BCh}$. Thus this existential requirement is satisfied if there exists a reachable marking starting from which there is an execution of linearization of PCh followed by an execution of BCh.
(4) Now suppose $\langle P, \varphi, BCh\rangle$ is a basic existential chart with a pre-condition. Then TP satisfies $\langle P, \varphi, BCh\rangle$ iff there exists a reachable marking M_0 and $M_0 \overset{\sigma_0}{\Longrightarrow} M_1$ such that $V_0 \models \varphi$ and a prefix of σ_0 corresponds to a member of $lin(BCh)$.
(5) Basic charts with post-conditions are dealt with similarly. For instance, suppose $[BCh, P, \varphi]$ is basic universal chart with a post-condition. Then TP satisfies this requirement iff every reachable marking M_0 satisfies the following condition. Suppose $M_0 \overset{\sigma_0}{\Longrightarrow} M_1$ such that $\lceil_{BCh} (\sigma_0)$ has a prefix which is a member of $lin(BCh)$ and σ is the least prefix of σ_0 with this property. Then $V \models \varphi$ where $M_0 \overset{\sigma}{\Longrightarrow} M$ and $M = (s, V)$. Thus whenever an execution

of BCh takes place then at the resulting marking, the condition φ holds. The semantics of the basic existential chart with a post-condition is defined in a similar way.

One can effectively decide whether or not a bounded CTP TP satisfies an LSC requirement lsc. This is so because from $\mathrm{PN}_{\mathrm{TP}}$ (the Petri Net corresponding to TP), we can extract a finite Kripke structure. Moreover, it is known that lsc can be effectively transformed into a CTL^* formula [14]. As a result we can apply the known model checking procedure for CTL^* to solve this problem [6]. This however will involve high computational complexity and more efficient decision procedures are needed to solve this problem.

6.2 Translating CTP models to LSC

We can also translate a CTP model into an LSC specification. Consequently the play engine mechanism developed in the LSC framework [15] becomes readily accessible for simulating CTP models. Furthermore, this translation also makes it clear that the CTP model is a restricted version of the LSC formalism in which only universal charts are used but the intra-object control flow is explicitly specified using traditional mechanisms.

Let $\mathrm{TP} = \{TS_p\}_{p \in \mathcal{P}}$ be a CTP over $(\Gamma, \mathcal{P})$ where Γ is a finite set of transaction schemes over $(\mathcal{P}, M, Act)$. Assume as before that each process $p \in \mathcal{P}$ is associated with $TS_p = (S_p, \Gamma_p, \longrightarrow_p, init_p, V_{p,in})$. Recall also that the set of pre and post control states of the transaction scheme T denoted as ${}^\bullet\mathrm{T}$ and $\mathrm{T}^\bullet$. To construct the LSC specification of TP, we will deploy $\bigcup\{S_p \mid p \in \mathrm{P}\}$ also as atomic propositions. Now let T be a transaction scheme of TP with $\mathrm{T} = \{[I^i : Ch^i : O^i] \mid i = 1, 2, \ldots, n\}$. We recall that each I^i is a propositional formula built out of AP, each $Ch^i = (E^i, \leq^i, \lambda^i)$ is a chart over $(\mathcal{P}, M, Act)$ and each O^i is a subset of AP. With T, we associate the LSC specification lsc_{T} given by $lsc_{\mathrm{T}} = lsc_1 \wedge lsc_2 \ldots \wedge lsc_n$ where for each i we have $lsc_i = [BCh_i, post_i]$ with $BCh_i = [pre_i, Ch^i]$. Also, pre_i and $post_i$ are given by $pre_i = \bigwedge_{s \in {}^\bullet\mathrm{T}} s \wedge I^i$ and $post_i = \bigwedge_{s \in \mathrm{T}^\bullet} s \wedge \bigwedge_{A \in O^i} A$.

Intuitively, T translates to the universal requirement: whenever the pre-control states of T hold and the guard for the i-th transaction holds, then the i-th transaction *must* execute followed by the holding of the post-valuation O^i and the post-control states of T. The actual semantics is given in an asynchronous execution framework.

7 Using the CTP language

Based on the abstract formal model presented so far, we have designed a simple language, called CTPL, in order to explore the feasibility of using our approach for system-level design of reactive embedded systems. In order to develop CTPL into a full-fledged modeling language, we have had to elaborate several features of the formalism such as (a) syntax/semantics of the internal actions, (b) data types of messages sent and received, (c) local variable declarations in individual processes etc. Here we shall touch upon the major issues. The full syntax of the current version of the language can be obtained from `www.comp.nus.edu.sg/~ctp`.

Language Features We use a simple imperative language without iterations to describe the internal actions. Thus an internal action is an imperative program with arithmetic and boolean expressions, whose control flow is acyclic. Note that this is not a restriction in the expressive power of the language, since the overall control flow of a process allows (potentially) unbounded iterations. A related issue is that our current modeling language does not allow iterative executions within a transaction scheme. Such an extension would allow one execution of a transaction scheme γ to be specified as a number of iterations of the constituent transactions (where in each iteration, one of the constituent transactions is executed). Exit from the execution of γ happens when the guards of none of the constituent transactions of γ are enabled. In future, we plan to extend CTP (and CTPL) along these lines to support the specification of iterations within a transaction scheme. This will of course naturally define iterations for internal actions as well, since an internal action is simply a degenerate transaction scheme. For the data types of messages as well as local variables of processes, the language implementation currently supports scalar types (such as boolean, integers, user-defined subrange types) as well as vectors (registers) of these scalar types. The guards of transactions are boolean expressions, where the propositions in the boolean expression are allowed to use arithmetic expressions.

Language Implementation Currently the implementation of CTPL is supported with the following tools and applications.

- a Graphical User Interface for constructing diagrammatic specifications and visualizing them
- a translator for converting visual CTP specifications to a textual format (based on XML)
- a scanner and parser for the textual format generated by translating the visual specifications
- A translator that produces Verilog code from the Intermediate Representation produced by the parser.
- Modeling and deriving of an FPGA-based implementation of an embedded controller for a 16-chamber micro-Polymerase Chain Reaction (μ-PCR) biochip. This real-life embedded controller co-ordinates a complex thermal cycling process requiring highly accurate temperature control and real-time temperature monitoring.

All of these tools are under active development and again, more details can be found at `www.comp.nus.edu.sg/~ctp`.

Integration into co-design environments One of our goals is also to integrate CTPL as a front-end into a hardware-software co-design toolkit. Towards this goal, we are working on translating CTPL into the Metropolis Meta Model

(MMM) language [3]. MMM is a common intermediate language for specifying heterogeneous embedded systems, and allows for simulation of system descriptions specified via different models of computation. The Metropolis project provides a SystemC based simulator for generating traces of a system described MMM; this is useful for functionality checking and performance evaluation. Based on experience gained through hand-translations of simple bus protocol examples from CTPL to MMM, we have developed a strategy for designing the translator. In the current version, we are implementing transaction schemes in CTPL via more centralized channels in MMM. The results concerning this effort will be reported elsewhere.

Formal Verification We are also working on automated verification of CTPL programs via model checking. Towards this end, we have developed a translator from CTP to the input language of the SMV[4] model checker. However, the state machine like input language of SMV has a very different specification style as compared to CTPL and hence the SMV-based verification method does not appear to be the ideal one for CTPL specifications. Consequently, we are also building a translator from CTP to Promela (the input language of the SPIN [18] model checker). Unlike the SMV model checker, Spin allows modeling of explicit control-flow within processes. This is similar in flavor to the process specification style of the CTP modeling language. Asynchronous message passing communication (as used in our MSCs) is also directly supported in Promela via channels.

Synthesis As for automatic synthesis, we have developed a translator to generate Verilog descriptions from CTPL programs, thereby creating a path to hardware implementation. So far, we have not studied the means for generating software code from CTPL specifications. One possibility would be to convert CTPL specifications to multi-threaded programs (where the processes in a CTPL specification map to threads in the generated program). This may require translating the message passing style of communication espoused in CTPL (via the use of MSCs) into shared variable communication among threads in the target programming language (such as Java). Developing an automated translation scheme for generating multi-threaded Java code from CTPL specifications is a topic of current and future work in our project [25].

8 Discussion

In this paper we have presented CTP, a high level specification language for modeling reactive systems. Our model is based on Message Sequence Charts (which emphasize inter-process communication) and explicit description of intra-process computations and control flow. The main questions to be pursued in this context involves well-formedness checking, formal verification as well synthesizing implementations from such models.

We have constructed a translator that transforms a CTP program into an internal representation of the Petri net representing the behavior of the CTP model. A crucial step in this translation consists of obtaining the event structure representation of each transaction scheme. Using this translator we are currently automating the analysis of transaction schemes for well-formedness. Work is also underway to devise a more efficient and direct procedure for determining the boundedness property. An interesting related problem is the issue of schedulability analysis as formulated in [24], which focuses on ensuring bounded message queues during system execution.

As for formal verification, the Petri net representation of bounded CTP models can be represented as a finite transition system. Indeed, due to the presence of the atomic propositions, this transition system can be viewed as a Kripke structure. Hence dynamic properties of the system being modeled as a CTP can be specified in a temporal logic such as LTL and formal verification of these specifications can be carried out using model checking tool such as Spin [18].

We are also currently exploring the means for extending our model along a number of dimensions, namely: parameterizing each component as an instance of a class together with the parameterization of the transaction schemes; further relaxing the control flow restrictions; and, adding timing constraints. Introduction of classes and objects into our model is particularly interesting since it can allow to symbolically simulate a CTP specification with unboundedly many objects (which form finite number of classes in terms of behaviors). Formalizing these ideas and observations is a topic of future work.

9 Acknowledgments

Preliminary versions of parts of this paper have appeared in [22] and [23]. We would like to thank the anonymous referees of these two papers for their comments. This research has been supported by an A*STAR (Agency for Science, Technology and Research, Singapore) Research Grant 022 106 0042 funded under the Embedded and Hybrid Systems Program.

We would like to thank Tran Tuan Anh for providing a substantial amount of inputs for all aspects of the research reported here. We would also like to thank the following people for their contributions in the implementation of the CTP language and its usage in system level design: Prakash Chandrasekaran, Kathy Nguyen Dang, Roman Gagarsky, Pankaj Jain, Nikhil Jain and Kamrul Hasan Talukder.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2001.
2. ARM Limited. *AMBA On-chip Bus Specification*, 1999.
3. F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, Y. Watanabe, and G. Yang. Concurrent execution semantics and sequential simulation algorithms for the Metropolis Meta-Model. In *International symposium on Hardware/software codesign (CODES)*, 2002.

4. Cadence Berkeley Laboratories, California, USA. *The SMV Model Checker*, 1999. `www-cad.eecs.berkeley.edu/~kenmcmil/smv/`.
5. B. Caillaud, P. Darondeau, L. Helouet, and G. Lesventes. Hmscs as partial specifications ... with pns as completions. In *Modeling and Verification of Parallel Processes 4th Summer School, MOVEP 2000, LNCS 2067*, 2001.
6. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.
7. Codesign, Simulation and Synthesis (COSY) project. *Generic Interface Modules for PI-Bus*, 2001.
8. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.
9. J. Desel and J. Esparza. *Free Choice Petri Nets*. Cambridge University Press, 1995.
10. B.P. Douglass. *Doing Hard Time: Developing Real-time Systems using UML, Objects, Frameworks and Patterns*. Addison-Wesley, 1999.
11. D.D. Gajski, J. Zhu, R. Dmer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.
12. T. Grotker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
13. D. Harel and H. Kugler. From play-in scenarios to code: An achievable dream. In *Fundamental Approaches to Software Engineering (FASE), LNCS 1783*, 2000.
14. D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *International Journal on Foundations of Computer Science*, 13(1), 2002.
15. D. Harel, H. Kugler, R. Marelly, and A. Pnueli. Smart play-out of behavioral requirements. In *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2002.
16. D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.
17. 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.
18. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
19. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Springer Verlag, 1997.
20. I. Krueger, R. Grosu, P. Scholz, and M. Broy. From MSCs to statecharts. In *International Workshop on Distributed and Parallel Embedded Systesms (DIPES)*, 1998.
21. M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains. *Theoretical Computer Science (TCS)*, 13, 1981.
22. A. Roychoudhury and P.S. Thiagarajan. Communicating transaction processes. In *IEEE International Conference on Applications of Concurrency in System Design (ACSD)*, 2003.
23. A. Roychoudhury and P.S. Thiagarajan. An executable specification language based on message sequence charts. In *Formal Methods at the Crossroads: from Panacea to Foundational Support*. Springer Verlag, LNCS 2757, 2003.
24. M. Sgroi and L. Lavagno. Synthesis of embedded software using free-choice Petri nets. In *ACM Design Automation Conference (DAC)*, 1999.
25. P.S. Thiagarajan et al. Communicating Transaction Processes (CTP) project, 2003. `http://www.comp.nus.edu.sg/~ctp`.
26. Z.120. Message Sequence Charts (MSC'96), 1996.