Honours Year Project Report

Analysis Methods for Cyclic Transaction Processes

by
Tran Tuan Anh

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2002/2003
Honours Year Project Report

Analysis Methods for Cyclic Transaction Processes

by

Tran Tuan Anh

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2002/2003

Project No: H06011
Advisor: Professor P.S. Thiagarajan
Deliverables:
  Report: 1 Volume
  Program: 1 CD
ABSTRACT

Message Sequence Charts (MSC) have been traditionally used to depict execution scenarios in the early stages of design cycle. MSCs describes intuitively inter-process interaction, but synthesizing intra-process executable specifications from an MSC-based description is a non-trivial task. Roychoudhury and Thiagarajan (2002) developed an executable formalism called Communicating Transaction Processes (CTP) which uses high-level transition systems to capture the control flow of the system components while using MSCs to describe the non-atomic component interactions. Because of the balance between control flow and component interactions, the CTP model is flexible, expressive, and at the same time amenable to formal analysis and synthesis. In this project, we explore in depth analysis and translation issues for the CTP model. We show how to construct a coloured Petri net from a CTP model, thus leading to a standard operational semantics. It allows us to generalize the basic concepts/analysis methods, also to use the existing tools. Regarding the translation issue, we develop a system which transforms a CTP model in textual format to an Intermediate Representation (IR) which can be used for translation into SMV, Verilog, and the like. Efficient algorithms for translation to IR, and checking properties of IR have been devised.

Subject Descriptors:
D.2.1 Requirements/Specifications
D.2.2 Design Tools and Techniques
D.2.4 Software/Program Verification

Keywords:
Formal Specification, Message Sequence Charts, Component Interfaces.

Implementation Software and Hardware:
ACKNOWLEDGMENTS

Many people have provided me a great deal of help to complete this project. My sincere apologies to whom I have forgotten to mention here.

I would like to thank my supervisor Professor P.S. Thiagarajan. He has been very helpful, and patient during the project. Without his valuable advice and guidance I would have not completed this project. He introduced me to the field of formal verification, and his work is a great source of inspiration.

I thank also my supervisors and lecturers – who inspired me to pursue a career in Computer Science – over the years. I mention particularly Dr. Nguyen Xuan My, his guidance during high school time is invaluable, he is a real character for whom I always admire.

Friends are always an important source of encouragement and joy. I am grateful to know them: Ha Thanh Trung, Dinh Thi Mai Lam, Nguyen Khuong Tuan, Dao Trong Chi and Dao Mai Anh, Le Hoang Dung, Nguyen Hoai Thu, Cung Thai Son, Ho Dieu Van to name a few. Special thanks to my girl-friend Tran Hong Ha, her understanding has turned this tough year into an enjoyable one.

Last but not least, thanks go to my family. I cannot thanks them enough for their help and support. They are always a tower of strength to me when I felt down.
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Title</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td></td>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>1</td>
<td>Chapter 1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Chapter 2. Preliminaries</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Petri Nets</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Coloured Petri Nets</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Message Sequence Charts</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Event Structures</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Chapter 3. The CTP Model</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>The Definition of CTP Model</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Chapter 4. The Semantics</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>From Transaction Schemes to Event Structures</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>The Coloured Petri Net Semantics</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>Chapter 5. Analysis and Verification</td>
<td>22</td>
</tr>
<tr>
<td>5.1</td>
<td>Well-formed Transaction Schemes</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>From Transaction Schemes to Event Structures</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Behaviour Properties</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>Chapter 6. Implementation</td>
<td>35</td>
</tr>
<tr>
<td>6.1</td>
<td>Overall System Design</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>Textual Input Format</td>
<td>36</td>
</tr>
<tr>
<td>6.3</td>
<td>Domain Classes</td>
<td>37</td>
</tr>
<tr>
<td>6.4</td>
<td>Algorithmic Classes</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>Chapter 7. Discussions</td>
<td>39</td>
</tr>
<tr>
<td>7.1</td>
<td>Summary of the project</td>
<td>39</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>41</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1. The producer/consumer example. .......................... 5
Figure 2.2. A Message Sequence Chart. .............................. 7
Figure 2.3. An Event Structure. ...................................... 9
Figure 3.1. Inter-process communication and intra-process control flow. 11
Figure 3.2. Choice of inter-process communication. .................. 12
Figure 3.3. Distributed nature of choice in a net transition. ....... 13
Figure 4.1. Event Structure for Transaction Scheme. ................ 18
Figure 4.2. The Acyclic Net. ......................................... 20
Figure 5.1. A Transaction Scheme which is not well-formed. ....... 23
Figure 5.2. The process-bus interaction example. .................... 30
Figure 5.3. The process-bus interaction example. .................... 31
Figure 5.4. The process-bus interaction example in CPNtool. ....... 32
Figure 5.5. Transitions in CPNtool. .................................. 33
Figure 6.1. The Overall System. ..................................... 35
Figure 6.2. The Visitor pattern. ..................................... 38
CHAPTER 1

INTRODUCTION

Message Sequence Charts (MSCs) – also known as timing sequence diagrams, message flow diagrams, or object interaction diagrams – are an appealing visual formalism used to document system requirements in the early design stages. MSCs are used in various software engineering methodologies and tools for concurrent, reactive, distributed systems (Booch, Jacobson, Rumbaugh, and Rumbaugh, 1998; Douglass, 1999; Selic, Gullekson, and Ward, 1994). In the simplest form, an MSC depicts the exchange of messages between the processes of the system, and corresponds to a single partial-ordered execution of the system. MSCs are used to document system requirements that guide the system design, describe test cases and scenarios, express system properties, etc. MSCs and a related mechanism called HMSCs (Hierarchical Message Sequence Charts) have been standardized by ITU (Z.120, 1996) for specifying telecommunication software, and a version of MSCs called Sequence Diagrams are incorporated in software engineering notations such as the UML standard (Booch et al., 1998; Douglass, 1999).

In all these settings, MSCs are used to capture system requirements. To move towards an implementation, we must obtain an executable specification which is related in some manner to the MSC-based requirements. The key difficulty here, as identified in (Harel and Kugler, 2001), 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 behaviours (say, one state-chart for each object). This is a difficult problem and it has been studied in various limited contexts (Alur, Etessami, and Yannakakis, 2001; Harel and Kugler, 2001; Henriksen, M.Mukund, Kumar, and Thiagarajan, 2000).

Roychoudhury and Thiagarajan (2002) propose a method of using MSCs to construct executable specifications in a more direct fashion. Their aim is to strike a balance between control flow and component interactions. The model is called Communication Transaction Processes (CTP), and it is flexible, expressive, and at the same time amenable to formal analysis and synthesis.

The main idea of the CTP model is to use traditional methods to capture the control flow of the system components while using MSCs to describe the non-atomic component
interactions. The well-known model of synchronized product of transition systems is chosen to describe the control flow. Synchronized product of transition systems in a multiple-component system is a network of labeled transition systems that synchronize on common actions. That is to say each component (agent) is presented as a transition system; the actions (state-to-state transitions) are labeled, and different components synchronize on common actions. However, we impose restrictions on the control flow, the most important one is that the choice as to which actions to take part in is made by the components in a local fashion. We next refine each common abstract action \( \gamma \) involving a set of agents (components) into a transaction scheme \( T_\gamma \). Basically, a transaction scheme is a set of MSCs which describe a set of possible scenarios of interactions between agents. Each MSCs in transaction scheme \( T_\gamma \) is guarded by a propositional formulas built out of atomic propositions. Thus, when a transition scheme (common actions) is to be executed is determined by the control flow in the high level project transition system. As to which transaction (MSCs) \( T_\gamma \) will be chosen to executed is determined by the guards. The atomic propositions of a guard will capture the properties of agents involved in that scenarios. A central feature of the CTP model is that both control flow and the evaluation of the guards – which then leads to the execution of a specific transaction with in a transaction scheme – are all done in a distributed and asynchronous manner.

The project. Based on the CTP model in (Roychoudhury and Thiagarajan, 2002), the goal of this project is to explore in depth following aspects of the model.

- **Analysis Issues.** The aim here is to derive practical methods for determining some important behavioural properties, such as boundedness, liveness, and the like. In this aspect, based on (Roychoudhury and Thiagarajan, 2002) we construct the operational semantics of the CTP model in terms of coloured Petri nets. From the operational semantics, we will be able to build a program which translates CTP models into coloured Petri nets; the resulting coloured Petri net representations can then be fed in the existing coloured Petri net tools to analysis and verify. In addition, we give a detailed plan for using CPNetool (CPN Group) to check behavioural properties of the CTP model.

- **Translation Issues.** In order to implement CTP descriptions, one needs to
develop translation mechanisms that will take as input a CTP model and generate code in some language that is executable on a chosen architecture. One main issue is to design a format for machine-recognizable CTP descriptions and the second issue is of course the translation into the target language. In this end, we build a program which convert the the CTP model in textual format into an intermediate representation (IR) format based on a data structure called Event Structures. The IR can then be passed to various modules for hardware synthesis, translation in SMV tool, and so forth. For the IR syntax we also derive a method to check for its wellformedness which is defined in (Roychoudhury and Thiagarajan, 2002).

Report organization. The rest of this report is organized as follows. In the next section, we briefly review the formal definitions of the notions used in this papers, namely, Petri nets, coloured Petri nets, MSCs, and Event Structures. In Chapter 3, we present the CTP model while illustrating its main features with simple examples. Chapter 4 provides the operational semantics of the CTP model in terms of coloured Petri nets. On Chapter 5, we discuss some important (behavioral/other) properties of the model and the techniques we use for verifying these. In Chapter 6, the implementation issues are discussed. Finally, Chapter 7 summarize the project and outline future research directions.
Chapter 2

Preliminaries

In this chapter, we present a brief introduction to background material. We fix here all the definitions, notations, and the like which will be used later. We will give the definitions for Petri nets and Message sequence charts (MSC), the two main ingredients of CTP model. The colored Petri nets (CPN) used to describe the CTP semantics and the event structures (ES), a major step in describing the CTP semantics, are also presented.

2.1 Petri Nets

The area of Petri nets was initiated by C.A. Petri in the early sixties (Petri, 1962). However, Petri nets now is actually a generic name for a whole class of net-based models. A Petri net models the behavioral aspects of distributed systems. A good thing about Petri nets is that it is a graphical notion and at the same time a precise mathematical notion. The combination of these two makes Petri nets an attractive model.

In this report, a class of Petri nets called Place/Transition Petri nets is used in one step during the construction of coloured Petri Nets semantics for the CTP model. We just use the term Petri nets instead of Place/Transition Petri nets, even though Petri nets is a generic name. We only give a brief introduction to Petri nets. See (Desel and Reisig) for complete presentation of the model.

A Petri net $N = (S, T, F)$ is constituted by a set $S$ of places, a set $T$ of transitions such that $S \cap T = \emptyset$, and set $F$ of directed arcs (flow relation), $F \subseteq (S \cup T) \times (S \cup T)$. satisfying $F \cap (S \times S) = F \cap (T \times T) = \emptyset$.

All places and transitions, $X = S \cup T$, are said to be elements of a Petri net. For each element $x \in X$, its pre-set (or input-set) is defined by $\bullet x = \{ y \mid (y, x) \in F \}$, and its post-set (or output-set) is defined by $x^* = \{ y \mid (x, y) \in F \}$. For $Y \subseteq X$, we write $\bullet Y = \bigcup_{x \in Y} \bullet x$, and $Y^* = \bigcup_{x \in Y} x^*$. The flow relation $F$ connects places to transitions, and via versa, but never between places nor transitions. Whenever we want to indicate the net under consideration, the notations $S_N, T_N, \bullet x_N$, etc. will be used.
A marking of a net $N$ is a mapping $m : S_N \rightarrow \mathbb{N}$ where $\mathbb{N} = \{0, 1, 2, \ldots \}$. A place is marked by a marking $m$ if $m(s) > 0$. A marked Petri net $N = (S, T, F, m)$ is a Petri net $(S, T, F)$ with a marking $m$.

Figure 2.1 shows an marked Petri net system that models the well-known producer/consumer example. Places are denoted by circles, and transition are denoted by rectangles. The arrowed arcs denote the flow relation, and the small dots inside places denote the current marking of the net.

![Diagram of a Petri net system](image)

**Figure 2.1.** The producer/consumer example.

### 2.2 Coloured Petri Nets

In this section, we introduce the coloured Petri nets (CPN) as we use it in a later chapter to express the semantics of the CTP model. The definition is based on (Jensen, 1981). Formally, coloured Petri nets is just as powerful as Petri nets, however, coloured Petri nets helps us present the semantics of CTP cleanly.

**Definition 1** (Coloured Petri Nets). A coloured Petri net is a 5-tuple $CPN = (S, T, F, K, M)$ where

- $S$ is a set of places.
- $T$ is a set of transitions such that $S \cap T = \emptyset$.
- $F$ is a set of directed arcs (flow relation), $F \subseteq (S \cup T) \times (S \cup T)$, satisfying $F \cap (S \times S) = F \cap (T \times T) = \emptyset$.
- $K$ is a finite set of colours.
\textbullet \ M \text{ is a finite set of modes. Suppose } t \in T, \text{ then } \mu \in M \text{ is a mode of } t \text{ if, and only if, } \mu : (t \cup t^*) \to \mathbb{K}. \text{ We will write } t_{\mu} \text{ to denote } (t, \mu) - \text{ the mode } \mu \text{ of transition } t.

Basically, the coloured Petri nets are just like Petri nets, but each place there is a colour set attached to it, and each transition there is a mode set attached to it. Therefore, each token in a place will be coloured, and each transition will have several \textit{modes} of firing.

2.2.1 Markings and Behaviour

States of a coloured Petri net are defined by its markings. State changes are caused by occurrences of transition modes. Next we present the definition of markings, then we define the dynamics (or behaviour) of coloured Petri nets.

A \textit{marking} of a coloured Petri net is a mapping } m : S \times \mathbb{K} \to \mathbb{N} \text{ where } \mathbb{N} = \{0, 1, 2 \ldots \}. \text{ That is to say in a marking a place can have a number of token (possibly infinite), and every token has a colour chosen from the colour set of that place. Notice that two tokens can have the same colour.}

We now turn to the definition of the dynamic behaviour of coloured Petri nets.

\textbf{Definition 2. Let the colour Petri net } CPN = (S, T, F, K, M).

\textbullet \ Let the mapping } m \text{ is a marking of } CPN, \text{ and the function } \mu \in K \text{ is a mode under transition } t \in T. \text{ Then mode } \mu \text{ of transition } t, t_{\mu}, \text{ is \textit{enabled} in } m \text{ if, and only if, } m(s, t_{\mu}(s)) > 0 \text{ for each } s \in t^*.

\textbullet \ Let the mappings } m, m' \text{ are markings of } CPN. \text{ Then mode } t_{\mu} \text{ fires from } m \text{ to } m' \text{ if } t \text{ is \textit{enabled} in } m, \text{ and the marking } m' \text{ is defined by}

\begin{equation}
m'(s, \kappa) = \begin{cases} 
m(s, \kappa) - 1 & \text{if } t_{\mu}(s) = \kappa \text{ and } s \in t^* - t^*, \\
m(s, \kappa) + 1 & \text{if } t_{\mu}(s) = \kappa \text{ and } s \in t^* - \kappa, \\
m(s, \kappa) & \text{otherwise}
\end{cases}
\end{equation}

A \textit{marked} coloured Petri net is a coloured Petri net with an \textit{initial marking}, denoted by \((S, T, F, K, M, M_{in})\), where \((S, T, F, K, M)\) is a coloured Petri net, and \(M_{in}\) is an \textit{initial marking}. Since we are only interested in dynamic behaviour from now on we just say a coloured Petri net to mean a marked coloured Petri net whenever the context is clear.
2.3 Message Sequence Charts

As introduced in previous chapter, MSCs have a major role in the CTP model. We now give the formal definition for MSCs. A sample MSC is shown in Figure 2.2. Vertical lines in the chart correspond to asynchronous processes. Arrows denote message send-receive between these processes. Arrows point from sending events to receipt events. We adopt the usual MSC convention that arrows can be drawn either horizontally or sloping downwards, but not upwards. Each arrow is labeled with a message identifier.

Next we give the formal definition of MSCs.

![Diagram of a Message Sequence Chart]

**Figure 2.2. A Message Sequence Chart.**

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

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

Definition 3 (Message Sequence Charts). An MSC (over \((P, M, Act)\)) is a finite \(\Sigma\)-labeled poset \(M = (E, \preceq, \lambda)\) which satisfies the following conditions:

1. Each \(R_p\) is a linear order.

2. For every \(p, q\) with \(p \neq q\), \(|E_{plq} = E_{q?p}|\).

3. Suppose \(e \in E_{q?a}\). Then \(|e \cap E_{q?a}| = |e \cap E_{q?a}|\).

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

Then \(m = m'\).

5. \(\succeq = (R_P \cup R_{Ch})^\ast\) where \(R_P = \bigcup_{v \in P} R_p\) and \(R_{Ch} = \bigcup_{c \in Ch} R_c\).

The first condition says that all the events that process takes part in are linearly ordered; each process is a sequential agent. The second condition says that there are no dangling communication edges in an MSC, the number of sent message must be equal to the number of received messages. The third condition says that messages must be sent before they can be received; the order of communication actions must be correct. Furthermore, the fourth condition requires that the message names must be correctly ordered. Lastly, the partial-order of an MSC is its visual order; deduced by linear orders of participating processes and the send-receive order of messages.

2.4 Event Structures

In order to provide an operational semantics for the CTP model in terms of coloured Petri net, an key step is to combine the different guarded transactions within a transaction scheme into a single entity, and we do that by using Event Structure (ES). An ES is a mechanism for representing a parallel set of executions (computation trees). Each of transaction in a transaction scheme is presented by a computation tree in ES. We first introduce the definition of ESs with a graphical illustration. Next, the informal explanation of executions represented by an event structure is given. Last, we define a precise transition system associated with ES. The material of this section is based on (Thiagarajan, 2002).
Definition 4 (Event Structures). An Event Structure is a triple \( ES = (E, \preceq, \#) \) where

- \( E \) is a set of events.
- \( \preceq \subseteq E \times E \) is a partial order (reflexive, transitive, and symmetric) called the causality relation.
- \( \# \subseteq E \times E \) is irreflexive and symmetric binary relation called the conflict relation.
- \( \# \) is inherited via \( \preceq \) in the following sense: Suppose \( e_1 \# e_2 \) and \( e_2 \preceq e_3 \). Then \( e_1 \# e_3 \).

In short, ES is a partial order of event occurrences augmented with a binary relation called the conflict relation. We only consider only finite ESs in our context, in other words, the set of events is finite. Figure 2.3 is an example of ESs.

![Figure 2.3. An Event Structure.](image)

We have introduced here some graphical conventions for depicting this event structure, and these conventions will be followed elsewhere too. First, not all members of the \( \preceq \)-relation have been shown. Selected pairs (these are not required to be minimum as well) have been represented by directed arrows between the concerned pairs. Hence, in our example \( 1 \preceq 3, 5 \preceq 2, \) and \( 4 \preceq 6 \) etc. The remaining members of \( \preceq \) are to be inferred using the reflexive and transitive properties of \( \preceq \) (causality relation). Thus \( 1 \preceq 5, 3 \preceq 6 \) etc.

Second, only minimum set of the \#-relation have been shown. In the above example \( 5\#4 \). The \#-relation are depicted by squiggly lines between the corresponding pair of events. Like the causality relation, other members of the set are to inferred by using the fact that \# is irreflexive, symmetric, and inherited via \( \preceq \). Thus, \( 4\#5, 5\#6 \) etc.
Informally speaking, $ES = (E, \preceq, \#)$ represents executions involving the single occurrences of the events in $E$. If $e \preceq e'$, then in any execution, $e'$ can occur only if $e$ has occurred already. If $e \# e'$ then in no execution can both $e$ and $e'$ occur. For example, if $e_1 \# e_2 \preceq e_3$, then $e_3$ can occur only if $e_2$ has occurred, hence $e_1$ must have not occurred. On the other hand, if $e_1$ has occurred, then $e_2$ cannot occur, therefore $e_3$ cannot occur. Consequently, $e_1$ and $e_3$ can never both occur in an execution, and this explains the axiom that $\#$ is inherited via $\prec$.

However, we can express precisely the executions represented by ESs by associating a transition system with a ES. We briefly present the transition here. Let $e \in E$, then $\downarrow e = \{ e' \mid e' \preceq e \}$, we say $\downarrow e$ is the down closure of $e$. The set of events $c \subseteq E$ is a configuration if, and only if, $c = \downarrow c$ and $(c \times c) \cap \# = \emptyset$. That is to say a configuration is a subset of events which are down-closed and conflict-free. Intuitively, a configuration is a set of events that have occurred so far along some execution. Clearly, if $e$ has occurred and $e' \prec e$, then $e'$ must have also occurred; furthermore, if $e$ has occurred and $e \# e'$, then $e'$ could not have occurred. Let $c$ be a configuration and $e \in E$. Then $e$ is enabled at $c$ if, and only if, $e \not\in c$ and $c \cup \{ e \}$ is a configuration.

Briefly, the transition system associated with ES is an automaton whose set of states is the set of configurations, the initial state is the empty configuration, there is a transition from state (transition) $c$ to $c'$ by an event $e$ if, and only if, $e$ is enabled at $c$ and $c' = c \cup \{ e \}$. We are not interested in the languages accepted by the automaton. Therefore, there is no final states here.

Finally, in our context of the CTP model, event structures with labels are used. Let $\Sigma$ be a set of labels. Then a $\Sigma$-label event structure is a structure $LES = (E, \preceq, \#, \lambda)$, where $ES$ is an event structure, and $\lambda : E \to \Sigma$ is a labeling function. That is all we need to know about event structures.
In this section, we present the CTP model. The content of this chapter is adopted from (Roychoudhury and Thiagarajan, 2002). We first describe the CTP model informally with the help of simple examples followed by a formal definition. Being based on MSCs, the 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 show in Figure 3.1. Each process interacts with the other process and then performs some internal actions; this is done repeatedly. Note that the inter-process interaction and the internal actions have been separated into distinct units. A number of processes $P$ might execute a chart. A process $p$ which takes part in this execution might next participate in the execution of a chart involving a different set of processes, say $Q$.

![Figure 3.1. Inter-process communication and intra-process control flow.](image)

The organization of interactions into the various units is up-to the convenience of the designer. For example, in Figure 3.1 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 3.1 is essentially a Petri net where the local control states in each process are the places of the net. 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 3.1 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, in case more than one chart is associated with a transition, is based on the value of the local variables of the processes. For example, in Figure 3.2 the choice of which MSC to execute is decided by the value of local variable \textit{free} in process \( b \). If \( b,\textit{free} \) holds once control reaches \( s_1, t_1 \) respectively, we must execute the left-hand chart in Figure 3.2

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{chart.png}
\caption{Choice of inter-process communication.}
\end{figure}

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 3.3. If \( p_1,\textit{data} \) 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 \textit{free} of process \( b \). As shown in Figure 3.3, 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 the sense that it will in general involve propositions belonging to different processes participating in the MSC. We shall impose no restrictions on the guards at this stage. At the end of the execution of a chart, the truth values of the various propositions will be set to new values in general. In the examples above, we have not specified this. We shall return to this point when we define the CTP model formally.
3.1 The Definition of CTP Model

The high level control flow can be captured by a variety of models of distributed systems. Here we shall use here a restricted class of product transition systems but which can be represented as 1-safe free choice nets.

A product transition system is a network of sequential 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 $\Gamma$ and a family $\{\Gamma_p\}_{p \in \mathcal{P}}$ with each $\Gamma_p$ a subset of $\Gamma$ and $\bigcup \Gamma_p = \Gamma$. This induces the function $\text{loc}$ which assigns to each abstract in $\Gamma$ the set of agents that participate in the execution of that action. This function is given by: $\text{loc}(\gamma) = \{p \mid \gamma \in \Gamma_p\}$. If $\text{loc}(\gamma) = \{p\}$ then $\gamma$ will be called $p$-local action. The members of $\Gamma$ will be treated as abstract action labels in the first step where were define the control flow model. In the second step the will be interpreted as transaction schemes and further elaborated. $\Gamma_p$ is the set of actions that the process $p$ will participate in.

Anticipating the need to build guards in the second steps 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 $AP_p$ as $AP$. Each subset of $AP_P$ will be called $P$-valuation. In case, $P = \{p\}$ is a singleton we will write $p$-valuation instead of $\{p\}$-valuation.
For each $p$ let $TS_p = (S_p, \Gamma_p, \rightarrow_p, init_p, V_{p, in})$ be a finite state transition system over $\Gamma_p$ 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, 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). Moreover, to avoid notational clutter, we will require that each member of $\Gamma_p$ is the label of at most one transition in $TS_p$. These two restrictions can be formalized as follows and we shall assume that each $TS_p$ satisfies these two restrictions.

1. If $s \xrightarrow{\gamma} p s_1$, $s \xrightarrow{\gamma'} p s_2$ (possibly $s_1 = s_2$), then $\gamma$ and $\gamma'$ are $p$-local actions. In other words, $loc(\gamma) = loc(\gamma') = \{p\}$.

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 5** (Product Transition Systems). A product transition system over $\{\Gamma_p, AP_p\}_{p \in P}$ is denoted as $\{TS_p\}_{p \in P}$ where each $TS_p$ is as specified before. This product transition system is defined to be the transition system with an initial valuation given by $(S, \Longrightarrow, init, V_{in})$ where

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

Next, we need to define transaction schemes.

**Definition 6** (Transaction Schemes). A Transaction Scheme is a finite collection of guarded Message Sequence Charts $\{[I^i : Ch^i]\}_{i=1}^k$. Each $Ch^i$ is an MSC over $(P, M, Act)$. 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 pre-condition. 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 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 7** (CTP System Model). A CTP is a product transition system $\{TS_p\}_{p \in \mathcal{P}}$ over $(\Gamma, \mathcal{P})$ where $\Gamma$ 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 $\gamma$ is viewed as an abstract action label in high level product transition system whereas $agents(\gamma)$ is the set of agents participating in some transaction associated with the transaction scheme $\gamma$. More precisely, let $\gamma = \{[[I^i : Ch^i : O^i]]\}_{i=1}^k$ with $Ch^i = (E^i, \leq^i, \lambda^i)$ for each $i$. Then $agents(Ch^i) = \{p \mid E_p^i \neq \emptyset\}$ and $agents(\gamma) = \bigcup_{i=1,...,n} agents(Ch^i)$. 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 is still a generous restriction in that the distribution of transactions across the various transaction schemes can be reorganized according to the needs of the designer.
CHAPTER 4
THE SEMANTICS

In this chapter, we provide an operational semantics for the CTP model in terms of coloured Petri nets. This coloured Petri net semantics is derived from the Petri net semantics in (Roychoudhury and Thiagarajan, 2002). By using coloured Petri nets, we are able to present the operational semantics neater. A key step in our semantics is to combine the different guarded transactions within a transaction scheme into a single entity. After that, the coloured Petri net is built upon. We will first present the translation from transaction schemes to event structures, and then the construction of coloured Petri nets from the CTP model.

4.1 From Transaction Schemes to Event Structures

We first recall some notations. Let $AP = \bigcup_{p \in P} AP_p$ is the set of atomic propositions. Also let $T$ be a transaction scheme of the form $T = \{[I^i : Ch^i : O^i]\}_{i=1}^n$ where each $I^i$ is a propositional formula built out of $AP$, each $Ch^i = (E^i, <^i, \lambda^i)$ is a chart over $(P, M, Act)$, and each $O^i$ is a subset of $AP$. We let $T^i = [I^i, Ch^i, O^i]$ for each $i$ and call $T^i$ 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 $\{E^i\}_{i=1,...,n}$ are pairwise disjoint.

We now show how to construct the labeled event structure $ES_T = (E, \preceq, \#, \lambda)$ to be associated with a transaction scheme $T$. The set of events $E$ is obtained from the event sets $E^i$ for $i = 1, 2 \ldots n$, but after identifying events having isomorphic pasts. We start with a set $X$ whose elements are of the form $(e, i, P, V_P)$ where element $e \in E^i$, the set of processes $P = \{p \mid \exists e' \in E^i_p \text{ and } e' \preceq^i e\}$, 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. Actually, the second and third components in $(e, i, P, V_P)$ are redundant in the senses that they can be deduced by first twos. However, we will carry them for convenience, and notice that $P$ is uniquely defined given $(e, i)$ but $V_P$ is not. Next, let $x = (e, i, P, V_P)$ and $y = (d, j, Q, V_Q)$ be elements in $X$. Then $x \equiv y$ if, and only if, the down-closure $\downarrow e$ in $Ch^i$ is isomorphic to the down-closure $\downarrow d$ in $Ch^j$ (the isomorphism involves both structures and labels of events). We denote the $\equiv$-equivalence class containing $x$ as $[x]$. 
Set of Events We now define $E$, the set of events of $ES_T$ to be $\equiv$-equivalence classes of $X$. In other words, $E = \{[x] \mid x \in X\}$.

Causality Relation Let $[x], [y]$ be in $E$. Then $[x] < [y]$ if, and only if, there exists $(e, i, P, V_P)$ in $[x]$ and $(d, i, Q, V_Q)$ in $[y]$ such that $e \preceq^i d$ and $V_P \cap AP_e = V_Q \cap AP_d$ for each $p \in P$.

Conflict Relation We shall define the conflict relation $\#$ in two steps. 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 exists $(e, i, P, V_P)$ 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 $\widehat{\#}$, $(b)$ inherits through causality, that is, $[x] \# [y]$ and $[y] \preceq [z]$ implies $[x] \# [z]$, and $(c)$ is a symmetric relation.

Labeling Function Finally, the labeling function $\Lambda$ is given by: $\Lambda([e, i, P, V_P]) = \lambda^i(e)$.

As a result, the tuple $ES_T = (E, \preceq, \#, \Lambda)$ is a labeled event structure (Roychoudhury and Thiagarajan, 2002). In Figure 4.1, part (a), (b) are two MSCs of a transition scheme, and part (c) is the corresponding event structure. We have just presented the event structure yielded by a transaction scheme. In Section 5.2, we will show a constructive method to build an event structure out of a transaction scheme, and we will prove that the constructive method complies with the event structures above.

4.2 The Coloured Petri Net Semantics

We will now construct the coloured Petri net semantics for the CTP model in three steps. First, we shall convert each labeled event structure yielded by a transaction scheme into a acyclic net. Next we will merge these nets with the high level control flow net. As a last step we will refine control states and the transitions to expose information about the valuations of the atomic propositions.
4.2.1 From Event Structure to Acyclic Net

First, let $T$ be a transaction scheme and $ES_T = (E, \preceq, \#, \lambda)$ be its event structure representation. For $e \in E$ we set $\text{proc}(e) = p$ if there exists $(x, i, P, V_P)$ in $e$ such that $x \in E^i_p$. It is easy to see that $\text{proc}$ is a well-defined function, and it is also easy to check that if $e \overset{\#}{\rightarrow} e'$, $\text{proc}(e) = \text{proc}(e')$.

Next, we define the relation $EQ^T_{\#} \subseteq E \times E$ as follows (where the context is clear, we will write $EQ$ instead of $EQ^T_{\#}$): $e \overset{\#}{\rightarrow} e'$ if, and only if, $e \overset{\#}{\rightarrow} e'$. Furthermore, if $e_1 \preceq e$, $e_1' \preceq e'$, and $e_1 \overset{\#}{\rightarrow} e_1'$, then $e_1 = e$ and $e_1' = e'$. The need for defining a new relation $EQ^T_{\#}$ is because the relation $\overset{\#}{\rightarrow}$ is not a minimal relation.

In the Figure 4.1, both $((\beta \| p, \text{no}), (\beta \| p, \text{yes}))$ and $((p \| \beta, \text{no}), (p \| \beta, \text{yes}))$ are $\overset{\#}{\rightarrow}$ but only the earlier is in $EQ$. We have the following facts (Roychoudhury and Thiagarajan, 2002):

- $EQ^T_{\#}$ is an equivalence relation.
- Suppose $e \prec e'$ and $\text{proc}(e) = \text{proc}(e')$. Then $e \preceq e''$ for every $e'' \in [e']_{EQ}$ where $[e']_{EQ} = \{e'' \mid e' EQ e''\}$.

Last, Let the minimal causality defined as $e \prec e'$ if, and only if, $e \preceq e'$, $e \neq e'$, and for every $e'' \in E$, $e \preceq e'' \preceq e'$ implies $e = e''$ or $e'' = e'$. We now define the net representation $N_T$ of $ES_T$ as the acyclic net $(B_T, E_T, F_T)$ where:

1. The set of transitions $E_T = E$. Recall that $ES_T = (E, \preceq, \#, \lambda)$. 

Figure 4.1. Event Structure for Transaction Scheme.
2. The set of places $B_T$ and the flow relation $F_T$ are the least sets which satisfy:

(a) Suppose $e < e'$ and $\text{proc}(e) \neq \text{proc}(e')$. Then $(e, e') \in B_T$, $(e, (e, e')) \in F_T$, and $((e, e'), e') \in F_T$.

(b) Suppose $e < e'$ and $\text{proc}(e) = \text{proc}(e')$. Then $(e, [e']_{EO}) \in B_T$ and $(e, (e, [e']_{EO})) \in F_{ES_T}$ and $((e, [e']_{EO}), e'') \in F_T$ for every $e'' \in [e']_{EO}$.

The set of transitions corresponds to the set of events in the event structure. Each edge in event structures will be presented by a places. However, we have to take into account the conflict relations. In the first case, the edge $(e, e')$ is cross-agent one, therefore, there is no conflict at event $e$ hence we just represent the edge as a place in the flow net. In the second case, all edges from $e$ to a set of conflict events will be represented by a single place. Figure 4.2 illustrates the acyclic net of the event structure from Figure 4.1. Notice we only take into account the minimal set of the causality relation to build the acyclic net.

### 4.2.2 Merging Acyclic Nets

Now we define the control flow Petri net $\text{CFN}_{TP}$ of the product transition system $TP$. Let us remind that $TP = \{TS_p\}_{p \in P}$ be a CTP over $(\Gamma, \mathcal{P})$ where $\Gamma$ is a finite set of transaction schemes over $(\mathcal{P}, \mathcal{M}, \mathcal{Axt})$. Let $TS_p = \{S_p, \Gamma_p, \rightarrow_p, init_p, V_{p, init}\}$ be the transaction system associated with transaction process $p$. For each transaction scheme $T$ in $\Gamma$ let $ES_T$ be its event structure representation and $N_T = (B_T, E_T, F_T)$, the net representation of $ES_T$. For convenience we will denote the set of pre and post control states of the transaction scheme $T$ as $\bullet T$ and $T^\bullet$ respectively and define these sets as: $\bullet T = \{s \mid T \text{ in } \Gamma_p \text{ for some } p, \text{ and } s \rightarrow_p s' \text{ for some } s, s' \in S_p\}$, similarly: $T^\bullet = \{s' \mid T \text{ in } \Gamma_p \text{ for some } p, \text{ and } s \rightarrow_p s' \text{ for some } s, s' \in S_p\}$.

We can now carry out the second step in providing the operational semantics. The control flow Petri net of $TP$ is the Petri net $\text{CFN}_{TP} = (S_{TP}, T_{TP}, F_{TP}, M_{in, TP})$ where:

- The set of places $S_{TP} = \bigcup\{S_p \mid p \in P\} \cup \{B_T \mid T \in \Gamma\}$.
- The set of transitions $T_{TP} = \bigcup\{E_T \mid T \in \Gamma\}$.
- The flow relations $F_{TP} = \bigcup_{T \in \Gamma} F_T \bigcup\{(s, e) \mid e \in \text{min}(E_T^p) \text{ and } s \in \bullet T \text{ and } s \in S_p\} \bigcup\{(e, s') \mid e \in \text{max}(E_T^p) \text{ and } s' \in T^\bullet \text{ and } s' \in S_p\}$.
\[ M_{\text{in}, TP}(z) = 1 \text{ if there exists } p \text{ such that } z = \text{init}_p \text{ (the initial state of } p) \text{. Otherwise, } M_{\text{in}, TP}(z) = 0. \]

Figure 4.2 is an example of an acyclic net linked to control states of the product transition system. By \( \min(E^o_T) \), \( \max(E^o_T) \) we mean the minimal (maximal) element of the set \( \{ e \mid \text{proc}(e) = p \} \) under the causality relation of the event structure \( ES_T \). The last step in our construction is to refine the places and the transitions of the control flow net to expose the valuations, and we use coloured Petri nets to do that.

![Diagram](image)

**Figure 4.2.** The Acyclic Net.

### 4.2.3 Constructing the Coloured Petri Net

Let \( CFN = (S_{TP}, T_{TP}, F_{TP}, M_{\text{in}, TP}) \) be the control flow net of a product transition system \( TP \), a communicating transaction processes \( CTP \). Then the coloured Petri net \( CPN_{TP} \) representing \( TP \) is \( CPN_{TP} = (S_{CPN}, T_{CPN}, F_{CPN}, K_{CPN}, M_{CPN}) \) satisfies following conditions:
• The set of places $S_{CPN} = S_{TP}$, the set of transitions $T_{CPN} = T_{TP}$, and the flow relation $F_{CPN} = F_{TP}$.

• Suppose $s$ is in $S_p$ where $TS_p = \{S_p, I_p, \rightarrow_p, init_p, V_p, in\}$. Then the colour $(\{p\}, V_{\{p\}}) \in K$ where $V_p$ is any $p$-valuation.

• Suppose $(e, b) \in F_T$ where $e \in E_{ES_r}$ and $b \in B_{ES_r}$. Then for each $(x, i, P, V_P) \in e$, the colour $(P, V_P) \in K$.

• Suppose $(e, b) \in F_T$ where $e \in E_{ES_r}$ and $b \in B_{ES_r}$. If there exist $(x, i, P, V_P) \in e$, and for each $s \in \bullet e$, there exist $(Q_s, V_{Q_s}) \in K$ such that $V_{Q_s} \cap AP_p = V_P \cap AP_p$ for each $p \in Q_s$. Then $t_p \in M$ defined by

$$t_p(s) = \begin{cases} (Q_s, V_{Q_s}) & \text{if } s \in \bullet t, \\ (P, V_P) & \text{if } s \in t^* \text{ and } s \notin S_p, \\ (\{p\}, V_{\{p\}}) & \text{if } s \in t^*, s \in S_p \text{ for some } p \text{, and } O^i \cap AP_p = V_{\{p\}}. \end{cases}$$

• $M_{in, TP}(init_p, \kappa) = 1$ if $\kappa = (\{p\}, V_{p, in})$.

The places, the transitions, and the flow relation of $CPN_{TP}$ are the same as those in the control flow net CFN. However, we capture the valuations by the colour set. For places representing states of transition system $TS_p$, the colours of those are just $p$-valuations, in other words, all the possible status of $p$. For places representing an edge in ESs, the colours will be all possible $P$-valuations that can be in those places.
A main goal of this project is to explore the analysis issues of the CTP model. In this chapter, we introduce some important properties of the CTP model and the techniques for determining these properties. First, we start with the notion of well-formed transaction schemes and the method to verify that. Next, we show how to construct explicitly the associated event structure of a transaction scheme as defined in Section 4.1. Last the issue of using existing coloured Petri net tools to check behavioural properties of the CTP model will be discussed in depth.

5.1 Well-formed Transaction Schemes

For pragmatic reasons, the definition of the CTP model in (Roychoudhury and Thiagarajan, 2002) imposes almost no syntactic restrictions. Consequently, one can easily specify behaviors which are problematic from both specification and implementation standpoints. For instance, consider the transaction scheme shown in Figure 5.1 and its associated event structure. If the control flow enables this transaction scheme and the valuation is \( \{ \neg A, B \} \), 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 \( \langle plq, m1 \rangle \) and \( \langle q'lp, m2 \rangle \) 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. In (Roychoudhury and Thiagarajan, 2002), the notion of well-formed transaction schemes is proposed as criterion for detecting and eliminating such undesirable behaviors. 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.

**Definition 8** (Well-formed Transaction Schemes). Let \( T = \{ T^i : \text{Ch}^i : \text{O}^i \}_{i=1,2,...,n} \) be a transaction scheme and \( ES_T = (E, \preceq, \Lambda) \) be its event structure representation. For a configuration \( c \) of \( ES_T \) we let \( ES_{c,T} \) be the sub-event structure induced by \( c \); it is the event structure \( (c, \preceq_c, \#_c) \) where \( \preceq_c \ (\#_c) \) is \( \preceq \ (\#) \) restricted to \( c \). Let \( \text{MAX}C_T \) be
the set of maximal configuration of $T$. Transaction scheme $T$ is said to be well-formed if, and only if, there exists a bijection $f : \{1, 2, \ldots, n\} \rightarrow MAXC_T$ such that $Ch^i$ is isomorphic to $f(i)$ for each $i$ in $\{1, 2, \ldots, n\}$.

Clearly the transaction scheme shown in Figure 5.1 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 an inconsistent specification at the level of transaction schemes can be caught. It should also be clear that well-formedness alone will not suffice guarantee sound implementations. For instance, if the behaviors exhibit non-determinism then hardware implementation can be problematic.

Each transaction scheme can be effectively analyzed to determine if it is well formed. Next, we present our algorithm to do so.

5.1.1 Transaction Scheme Well-formedness Checking Algorithm

Informally, the Definition 8 says that given a transaction scheme $T$, the associated event structure $ES_T$ is said to be well-formed if, and only if, the maximal executions of that event structure are precisely the executions of the transactions mentioned in the transaction scheme. Thus, the definition suggests that to check well-formedness of a transaction scheme we go through all maximal configurations of the associated event structure, and checks if each maximal configuration uniquely corresponds to a transaction in the transaction scheme. In addition, by making use of Lemma 1 below, we are able to traverse the set of maximal configurations effectively. Hence, the key idea for our well-formedness check algorithm is based on the Lemma 1. Basically, it says for a maximal configuration of an associated event structure, the maximal event of each
process $p$ (the last event $p$ executes) must correspond to a maximal event of the process $p$ in some MSCs. We say $p$-events to mention the set $e | \text{proc}(e) = p$.

**Lemma 1.** Let $T = \{ T^i = |I^i : Ch^i : O^i| \}_{i=1,2,...,n}$ be a transaction scheme, and $ES_T = (E, \leq, \# , \Lambda )$ be its event structure representation. Then for each maximal configuration $c$, if $e_p \in c$ is maximal event under $\leq^p$ (the $\leq$ relation restricted in $p$-events) in $ES_T$ for some $p$, then there exists an event $x \in Ch_i$ such that $x \in e_p$ (or $|x| \equiv e_p$) and $x$ is maximal event under $<^p$ in $Ch_i$.

**Proof.** We prove this lemma by using contradiction. First we define a set $MAX_p = \{ e | e \in ES_T$ and $e \equiv [x]$ where $x$ is a maximal event under $\leq^p$ in $Ch^i$ for some $i\}$. Let $c$ be a configuration of event structure $ES_T$, such that $c$ is maximal. Assume there exist a event $e_p \in c$ of process $p$, there not exist an event $e \in c$ such that $(e_p, e) \in \leq$, and $e_p \not\in MAX_p$. Hence, there must be an event $e'_p \not\in c$ such that $(e_p, e'_p) \in \leq$. Now consider the set $c' = c \cup \{e'\}$. Since $c$ is a maximal configuration, $c'$ can not be an configuration too. Therefore, event $e'$ must be in conflict with other event in $c$. Since $ES_T$ is the event structure associated with a set of MSCs, it is easy to see that the set of $p$-events in $ES_T$ has a total order. For that reason, $e'_p$ could not be in conflict with any $p$-events. Let say $e'_p \not\# e_q$ for some $q \neq p$. We also know that in an associated event structure, the minimal conflict is only between two events of the same process. Thus $e'_p \not\# e_q$, that means $e'_p$ is not in minimal conflict with $e_q$. As a result, there exist $e'_q \not\# e_q$ and $e'_p < e'_q$. Obviously, $e'_p \in c$, that is contradictory to the assumption that $c$ is a configuration. \square

The Lemma 1 leads to a simple algorithm for checking well-formedness of an transaction scheme. Since for each process $p$ in a maximal configuration of $ES_T$, the maximal events of $p$ in that configuration must be the corresponding class of the last event of $p$ in some MSC. Therefore, each process $p$ in a maximal configuration can only be in certain events which is the set of all corresponding classes of all maximal $p$-events across all MSCs. Hence, we assign a choice for the last event of each process $p$ from those sets, then from those chosen events we induce a maximal configuration. In other words, in order to traverse the set of maximal configurations, we just go through all possible way of assigning a last event to each process $p$. Next let us look at the algorithm in greater details.
Well-formedness check pseudo-code The algorithm for checking well-formedness of a transaction scheme is presented in Algorithm 5.1.

**Algorithm 5.1 WELL-FORMEDNESS-CHECK($ES_T$).**

**Input:** The event structure $ES_T = \{E, \preceq, \#, \Lambda\}$.

**Output:** Answer whether $ES_T$ is well-formed.

1. **for all** process $p$ **do**
   2. Create $MAX_p = \{e \mid e \equiv [x] \text{ where } x \text{ is maximal event under } \preceq^p \text{ in } Ch^i \text{ for some } i\}$.
3. **end for**

4. **PROPAGATECONFLICTRELATIONS($ES_T$),**
   {Generate full conflict relations in $ES_T$.}
5. **for all** maximal set $s \subset \bigcup_p MAX_p$ such that $s$ is conflict-free **do**
6. Check that each $MAX_p$ has only one element in $s$. If not **return** false.
7. Check if the configuration $c$ induced by $s$ is maximal one (we prove later that $c$ is the down-closure of $s$). To do so, we try to extend the configuration by adding new event. If not then **return** false.
8. Let $e_p$ is the event of process $p$ in $c$, let $H_p$ is the set of MSCs which have event belong to $e_p$. Check if $\bigcap_p H_p$ is singleton. If the set is not singleton then **return** false. If it is singleton then check if the corresponding MSC has been marked. If marked then **return** false.
9. **end for**
10. Check if all MSCs haven been marked. If all marked **return** true, else **return** false.

We describe more precisely each step of the Algorithm WELL-FORMEDNESS-CHECK. At the line 4, we generate full conflict relations by propagate down from the existing relations (remind that while building $ES_T$ the set of minimal conflict relations inserted).

The details of **PROPAGATECONFLICTRELATIONS** is as follows:

**Procedure** PROPAGATE-CONFLICT-RELATION($ES_T$)

**Input:** The event structure $ES_T = \{E, \preceq, \#, \Lambda\}$.

**Output:** Insert full conflict relations in $ES_T$.

1. Initialize an list $\mathcal{L} \leftarrow \text{PARTIALORDERSORT}(E)$.
2. **while** $\mathcal{L}$ is not empty **do**
   3. Remove the first event $e$ in list $\mathcal{L}$.
   4. For each event $e'$ such that $(e, e') \in \preceq$, add the conflict relation $(e', e'')$ in $\#$, if $(e, e'') \in \#$.
5. **end while**

It is easy to see that the procedure **PROPAGATECONFLICTRELATION** runs in $O(n^3)$ time where $n$ is the number of events in $ES_T$. The amount of storage used is in order of the number of conflict relations which might be exponential. We can prove the correctness of the procedure by loop invariant technique. The procedure maintains the
following invariant, which holds at any time: all conflict relations of visited events are inserted in $ES_T$. This can be proved by induction.

Next, how to go through all the maximal subsets in line 5 depends on the representation, but it is easy to do so. Other steps in WELFORMEDCHECK clearly explain themselves.

**Correctness** It remains to show that the algorithm WELFORMEDCHECK indeed returns the correct answer. The proof of that naturally follows the next lemma.

**Lemma 2.** The set of configurations created during the run of Algorithm WELFORMEDCHECK include all the maximal configurations of event structure $ES_T$.

**Proof.** First we prove that given a conflict-free subset $s$ of $\bigcup MAX_p$, its induced down-closure $c$ is a configuration. That because $s$ is conflict free hence its down-closure $c$ is conflict-free as well. Next, in each maximal configuration, events of each process $p$ has a total order, as a result, each $p$ has only one maximal event. By the Lemma 1 the maximal event of each process $p$ in a maximal configuration is from $MAX_p$, hence that configuration will be created during the run of WELFORMEDCHECK. $\square$

**Running time** There are too many factors affecting the size of the input, for example, the number of MSCs, the number of processes, etc. Hence, we do not intend to give a detailed analysis here. In this case, if $n$ is the number of events in all MSCs, $m$ is the size of conflict relations, and $k$ is the number of maximal configurations of $ES_T$, then the running time is in the order of $O(n + m + k)$. Informally, the factor involving $k$ is linear.

### 5.2 From Transaction Schemes to Event Structures

A major step in most of analysis and verification methods for the CTP model is to transform a CTP to an intermediate representation. In this case, we use Event Structures (ESs) to represent transaction schemes. The *mathematical* transformation (Roychoudhury and Thiagarajan, 2002) has already presented in Section 4.1, we now present an *algorithmic* method to convert a transaction scheme to an event structure.

We first recall some notations. Let $AP = \bigcup_{p \in P} AP_p$ is the set of atomic propositions. Also let $T$ be a *transaction scheme* of the form $T = \{|P^i : Ch^i : O^i|\}_i^{n-1}$, where each
$I^i$ is a propositional formula built out of $AP$, each $Ch^i = (E^i, \preceq^i, \lambda^i)$ is a chart over $(P, M, Act)$, and each $O^i$ is a subset of $AP$. We let $T^i = [I^i, Ch^i, O^i]$ 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 $\{E_i\}_{i=1, ..., n}$ are pairwise disjoint.

As defined in Section 4.1, the set of events $E$ is defined as $\equiv$-equivalent classes of events in $\bigcup Ch^i_{i=1, ..., n}$ (equivalent classes of isomorphic events in MSCs). Thus, let $x = (e, i, P, V_P)$ and $y = (d, j, Q, V_Q)$, then $x \equiv y$ if, and only if, the down-closure $\downarrow e$ in $Ch^i$ is isomorphic to the down-closure $\downarrow d$ in $Ch^j$. Since the down-closure of an event involves only predecessor events, hence we are able to define isomorphic events in a different way as follows.

**Definition 9 (Isomorphism of Events).** Let event $e \in E^i$ and event $d \in E^j$. Furthermore, let $\Pr(e) = \{g | g \preceq e\}$. Then $e$ and $d$ are isomorphic to each others, written as $e \equiv d$ if, and only if, the label $\lambda(e)$ is the same with the label $\lambda(d)$, and either

1. $e$ and $d$ are minimal events (they have no precedent events), or
2. there is a bijection $f : \Pr(e) \rightarrow \Pr(d)$, such that $g$ and $f(g)$ are isomorphic for each $g \in \Pr(e)$.

It is easy to see that the above definition is the same with the previous one. Clearly, each event $e \in E^i$ must correspond to an event in $ES_T$ which is $[e]$, hence intuitively (also informally) in order to construct $ES_T$ we have to create all the $\equiv$-classes and decide to which $\equiv$-class an event in MSCs belong. In the view of the fact that the new definition of isomorphism is recursive in nature, it suggests us that we construct the associated event structure $ES_T$ “recursively”. In other words, if we already have $[g]$ for all $g \in \Pr(e)$, then we are able to create $[e]$.

In brief, we will traverse events in transaction scheme $T$ according to $\preceq$ relations, in that way an event is visited only after all predecessor events. When we visit an event $e$, we will able to construct $[e] \in ES_T$ because we have all $\equiv$-classes of $\Pr(e)$ created.

**Transaction Schemes to Event Structures pseudo-code** Next, we look at the algorithm for constructing the associated event structure of a transaction scheme in more detail. The pseudo-code of the algorithm is shown at Algorithm 5.2.
Algorithm 5.2 \textsc{TransactionSchemeToEventStructure}(T).

\textbf{Input:} The transaction scheme \(T = \{|I^i: Ch^i: O^i|\}_{i=1,...,n}\).

\textbf{Output:} The associated event structure \(ES_T\).

1: \textbf{for all} \(Ch^i \in T\) where \(i = 1,...,n\) \textbf{do}
2: \hspace{1em} \(L \leftarrow \text{PartialOrderSort}(E^i)\)
3: \hspace{1em} \textbf{while} \(L\) is not empty \textbf{do}
4: \hspace{2em} \text{Remove the first event} \(e\) in the list \(L\). Event \(e \in Ch^i\) for some \(i\).
5: \hspace{2em} \text{Let} \(e' \in Ch^i\) is arbitrary event in \(Pr(e)\). Let \(p' \in ES_T\) is \([e']\) (there must exist such \(p'\)).
6: \hspace{2em} \textbf{for all} \(p \in \text{Child}(p')\) \textbf{do}
7: \hspace{3em} \text{if} \(e\) and \(p\) are isomorphic (Definition 9) \textbf{then}
8: \hspace{4em} \text{Make} \(p\) be the corresponding event of \(e\).
9: \hspace{3em} \textbf{end if}
10: \hspace{2em} \textbf{end for}
11: \hspace{1em} \textbf{if} there is no existing isomorphic event \textbf{then}
12: \hspace{2em} \text{Create a new event} \([e]\) in \(ES_T\).
13: \hspace{2em} \text{Add the causality relations to} \([e]\). For each event in \(Pr(e)\), make the isomorphic event in \(ES_T\) be the predecessor of \([e]\).
14: \hspace{2em} \text{Add the conflict relations to} \([e]\). All events which are not in the \(\downarrow [e]\), and belong to the same process are in conflict with \([e]\).
15: \hspace{1em} \textbf{end if}
16: \hspace{1em} \textbf{end while}
17: \textbf{end for}

\textbf{Correctness} The Definition 9 states that the isomorphism of events in MSCs and ES can be decided by only looking at the events themselves and the immediate predecessors. In addition, the procedure \textsc{TransactionSchemeToEventStructure} traverse events in MSCs according to the causality relation. Therefore, the correctness of \textsc{TransactionSchemeToEventStructure} naturally follows.

\textbf{Running Time} There are many parameters in our case. However, if we let \(n\) be the number of events in all \(MSCs\), and fix others to be constant factors, then we will have an \(O(n)\) running time.

5.3 \textbf{Behaviour Properties}

Via the coloured Petri net semantics in Section 4.2, the notions of a CTP being \textit{bounded}, \textit{live}, and \textit{dead-lock free} can be defined clearly. With our current system, we are able to transform a CTP to a coloured Petri net (CPN); the resulting CPN can be used as input for existing coloured Petri net analysis tools, such as CPN\texttt{tool} (CPN Group). Due to the time constraint, we have not implemented the transformation in our system. However, in
this section, through an example we will show precisely how to use an existing coloured Petri net tool, CPNtool, to verify properties of the CTP model. Notice that CPNtool is the name of a particular tool not a term describing a class of coloured Petri net tools, it is indeed a general term which might easily confuse readers.

Theoretically, it is easy to use existing tools, since we have already had the coloured Petri net semantics of the CTP model. We only have to describe the corresponding coloured Petri net of a CTP model in an analysis tool. Nevertheless, it is troublesome to do so in practice. The problem with using existing coloured Petri net tools – in this case CPNtool – is that the definition of coloured Petri nets used in the tools is different from ours. Hence, we have to find a way to express our semantics by using a different coloured Petri net definition. That is the main issue which we will demonstrate in this section.

In the following parts, we first present our CTP example, it is a simple CTP model describing data transaction between a process and a bus. We make a few simplifying assumptions about the example. Next, we discuss about the CPNtool, especially the definition of coloured Petri nets used by it. Last, the resulting coloured Petri net, which is built in CPNtool, of our CTP example is shown along with comments on transformation issues.

5.3.1 CTP process-bus interaction example

The process-bus interaction example is shown in Figure 5.2. There is a main transaction scheme involved both the process and the bus. There are transactions in this transaction scheme. In the first transaction, the process requests to send data, the bus refuses because it is busy; the transaction terminates. In the second transaction, the process requests to send data, the bus is free. Then the bus signals back, and the data transfer starts. We assume the process always has data to send to the bus, hence the leftmost transaction scheme, in which only p participates, simply is the internal computation of the process p. We also assume the bus will be free and be not free alternatively, hence the rightmost transaction scheme basically negates the value of variable b.free (the status of the bus). The example is simple, however it shows indeed some important issues, namely, the choices between different MSCs in a transaction scheme, and the change of valuations of processes.
5.3.2 Coloured Petri nets in CPNtool

We now have a look at the definition of coloured Petri nets used in CPNtool. Basically, the definition of coloured Petri nets in CPNtool (CPNtool-CPN) is similar to our definition (CTP-CPN). However, CPNtool-CPN is more generic, and has some certain features to make it easier to use in practice. In short, a CPNtool-CPN has a set of places, a set of transitions, a flow relation. Each place has a colour set attached to it. They the possible colours which tokens at a place could have. A marking assigns to each place a set of coloured tokens. The only difference (also the most important difference) of CPNtool-CPN from our CTP-CPN is how transitions fire – or how to change states of system.

In CTP-CPN, each transition has a set of modes. A mode is a function from the neighbourhood to the colour set; it tells what coloured tokens in input places be like so that the mode is enabled, and what tokens the mode will produce to output places if it fires. Formally, the mode \( \mu \) of transition \( t \) is a function \( \mu : (t \cup \bar{t}) \rightarrow \mathcal{K} \). It is a straightforward way, and is suitable for our need. However, CPNtool uses a different approach (Jensen, 1997). Each transition has a guard which is an expression of type boolean, each arc has an arc expression whose evaluation must yield a multi-set over the colour set that is attached to the corresponding place. In other words, each arc expression evaluation will produce a set of coloured tokens, and two or more tokens can have the same colour. It will be clear when we discuss about the behaviour of transitions.
In CPNtool-CPN, all the expressions are written in CPN ML (CPN Group), a slightly modified version of SML (Milner, Tofte, and Harper, 1990). Therefore, the colours is expressed by programming *types* rather than abstract sets.

\[
\text{if } x = p \text{ then } 1 \text{e else empty} \\
\text{(x) (y)} \\
\downarrow \quad \downarrow \\
\text{[fun}(x, y)\text{]} \\
\downarrow \\
\{3'x\}
\]

**Figure 5.3.** The process-bus interaction example

Having introduced the structure of CPNtool-CPN, now we will briefly go through the behaviour. In Figure 5.3, an example of a transition in CPNtool-CPN is shown. Because *guards* and *arc expressions* are written in CPN ML, they comprise of variables, operators, constructors, etc. To evaluate an expression you need first to *bind* a value to each variable, then compute the bound expression. For instance, in Figure 5.3 we have an arc expression “if \( x = p \) then \( 1 \text{e else empty} \)” where \( x \) is a variable, \( p \) are constant, \( 1 \text{e} \) means a set of one constant \( e \), and \text{empty} denotes a empty set. In this case, if we have a *binding* that binds \( x \) to constant \( q \), then the result of the expression will be an empty set. On the other hand, if we have a *binding* that binds \( x \) to constant \( p \), then result will be one token \( e \). A transition in CPNtool-CPN is *enabled* if there exist an *binding* such that the guard of that transition evaluates to true value, each input place has enough tokens generated by the evaluation of input arc expression. Clearly, the tokens generated by evaluations of arc expressions must have the types of corresponding places. For more detailed explanation, please see (Jensen, 1997).

### 5.3.3 Expressing the CTP semantics in CPNtool

Now we are ready to look at the resulting coloured Petri net of the example in CPNtool. Figure 5.4 is a snapshot of the example we *manually* built in CPNtool. As stated above, the structure is exactly the same as our CTP-CPN semantics. However, we have to use CPN ML to express colours, valuations, guards, etc.
Next, we will briefly go through main issues on transforming a model in CTP-CPN to CPNtool-CPN.

- **The structure.** The places, the transitions, and the flow relation in CPNtool-CPN are the same with CTP-CPN.

- **Local variables.** This is straightforward, since CPNtool support the use of variables. A variable in CPN ML can have all the common types, such as integers, reals, sets, etc. For instance, the variable free of the bus b in our example above is declared as “var bus free : BOOL.”

- **Valuations.** In CTP semantics defined in Section 4.2, the colours of tokens at a place is actually the $P$-valuations might reach that place. In CPN ML, we can present a valuation of a set of variables $\{v_1, v_2, \ldots, v_n\}$ as a value of the type “product $T_1 \ast T_2 \ast \ldots \ast T_n$” where $Ti$ is the type of $v_i$.

- **Transitions.** In CPNtool, we express firing modes as follows. Each transition has a guard function, each input arc expression is simply an variable, and each
output arc expression is a function. For example, we have a transition $t$, and there are $k$ modes associated with $t$, namely, $\{\mu_1, \mu_2, \ldots, \mu_k\}$. Transition $t$ has $n$ input places, and $m$ output places. Hence, in CPNtool, each input arc expression of $t$ is $\text{in}_i$, each output arc expression of $t$ is function $\text{out}_j(\text{in}_1, \text{in}_2, \ldots, \text{in}_n)$, and the guard function is $\text{check}(\text{in}_1, \text{in}_2, \ldots, \text{in}_n)$. Figure 5.5 illustrates the implementation of transitions in CPNtool, it also shows the skeleton of guard function $\text{check}(\ldots)$ and output functions $\text{out}_j(\ldots)$. Basically, the tuple $\text{cond}_i$ in both function $\text{check}(\ldots)$ and $\text{out}_j(\ldots)$ is the condition of the mode $\mu_i$ to be enabled. It is clear that the implementation described here expresses precisely the meaning of modes $\{\mu_1, \mu_2, \ldots, \mu_k\}$.

We now present the details of colour sets, variables, and functions that are used for our example.

**Listing 5.1. Colour declaration in CPNtool**

```plaintext
1 color BOOL = bool;
2 color INT = int;
3 color PROCESS.STATE = unit with e:
4 color BUS.STATE = BOOL;
5 color PROCESS.BUS.STATE = product PROCESS.STATE * BUS.STATE;
6 color STATE.1 = subset PROCESS.STATE with [e];
7 color STATE.2 = subset PROCESS.BUS.STATE with [(e, false), (e, true)];
8 color STATE.3 = subset PROCESS.BUS.STATE with [(e, false)];
9 color STATE.4 = subset PROCESS.BUS.STATE with [(e, true)];
10 color STATE.5 = subset PROCESS.BUS.STATE with [(e, true)];
11 color STATE.6 = subset PROCESS.BUS.STATE with [(e, true)];
```

**Listing 5.2. Variable declaration in CPNtool**

```plaintext
1 var bus.free : BOOL;
2 var in.bus : BOOL;
```
var in.pro : PROCESS.STATE;
var in.1 : PROCESS.STATE;
var in.2.a : STATE1;
var in.2.b : BUS.STATE;
var in.3 : STATE.2;
var in.4 : STATE.2;
var in.5 : STATE.4;
var in.6 : STATE.5;
var in.7 : STATE.6;
var in.8 : STATE.3;

### Listing 5.3. Function declaration in CPNtool

```plaintext
fun bus_1(x) = if x=true then 1 else false;  
fun bus_2(x) = if x=false then 1 else empty;  
fun check_bus_1(x) = if x=true then true else false;  
fun check_bus_2(x) = if x=false then true else false;  
fun process_1(x) = x;  
fun out_1(x) = if x = e then e else ();  
fun check_1(x) = if x = e then true else false;  
fun check_2(x, y) = case (x,y) of (e, true) => true | (e, false) => true | _ => false;  
fun out_2(x, y) = case (x, y) of (e, false) => (e, false) | (e, true) => (e, true);  
fun check_3(x) = case x of (e, true) => true | _ => false;  
fun out_3_a(x) = x;  
fun out_3_b(x, y) = y;  
fun out_8(x) = e;  
fun check_8(x) = case x of (e, false) => true | _ => false;  
fun out_5(x) = x;  
fun check_5(x) = case x of (e, true) => true | _ => false;  
fun check_6(x) = case x of (e, true) => true | _ => false;  
fun out_6_a(x) = e;  
fun out_6_b(x) = x;  
fun out_7(x, y) = y;  
fun check_7(x) = case x of (e, true) => true | _ => false;
```

It should be convincing now that we are able to translate a CTP model to a coloured Petri net in CPNtool. With our current system (described in next chapter), it is easy to automate that procedure.
A major part of this project is to build a system to facilitate the use of the CTP model. In this chapter, we shall present the overall design of the system, the textual input format, domain classes, and analysis classes.

### 6.1 Overall System Design

There is already an textual grammar for describing the CTP model which is described in next section. However, the current system is actually dealing with the Cyclic Transaction Processes model rather than Communicating Transaction Processes model. The only difference between those two is that in the Cyclic Transaction Processes model each process needs to be cyclic – a process executes repeatedly a list of transaction schemes. Nevertheless, it is easy to upgrade the system to accommodate the Communicating Transaction Processes model.

The overview of the whole system is described in Figure 6.1. First, the textual input describing a CTP model is scanned and parsed into data structures by using domain classes. Next, the translator will transform the input to an intermediate representation (IR). Last, IR will be passed to other components to perform requested actions, such as, well-formedness checking, liveness checking, translation to SMV, etc.

![Diagram](image)

**Figure 6.1.** The Overall System.

The system has been designed to be modular. A new analysis method can be programmed and plug in without effecting any other part of the system, we will discuss in more detail later. The language used in implementation is Java. In both architecture de-
sign and coding details of the system, we intensively use object-oriented technology. For instance, we use Visitor design pattern (E.Gamma, R.Helm, R.Johnson, and J.Vlissides, 1995) to define new operations performed on the intermediate representation, other design patterns used are Composite, Singleton, etc.

The codes of the system are organized into three packages:

- **ctp/io** includes all classes for input/output operations. The scanner and parser are in here.

- **ctp/ctpsdr** includes all domain classes.

- **ctp/analyser** includes all the analysis method classes.

### 6.2 Textual Input Format

The CTP model has a textual input format. In this section, we shall briefly summarize the syntax with a simple example, however we will not show the full grammar here. The following input example describes a CTP model of producer/consumer problem.

```plaintext
Listing 6.1. Textual input of a CTP model

ctp example{
  // Process Description Section
  process producer{
    0..15 : data_buf;  // local variables
    0..15 : addr_buf;
    boolean : ready;
    equation T1;  // control flow equation
  }

  process middle{
    ...  // similarly
  }

  process consumer{
    ...
  }

  // Scheme Description Section
  scheme T1{
    transaction send_data{
      agents{
        producer;  // agent
        recv(middle,claim_data), // event
        send(ack,1,boolean),
        send(addr,addr_buf),
      }
    }
  }
}
```
send(data, data_buf);
middle:
send(claim_data, 1, boolean);
recv(producer.ack),
{status:=din;},
recv(producer.addr),
recv(producer.data):
}
guard producer.ready & middle.status; // guard built out of local variables
}

transaction not.send.data{
.... // similarly
}
}

scheme T2{
...
// similarly
}

Basically the input consists of two section, Process Description section, and Transaction Scheme Description section. The first section, as the name suggested, describes all the processes taking part in the model. For each process p, information about its control flow, local variables and their type are provided. The transaction scheme section lists the transaction schemes mentioned in the process description section. Each transaction scheme is described as a set of guarded transactions. A guard of a transaction is a formula built on variables of processes involved. A transaction is a Message Sequence Chart, hence we list the sequence of events in each life-line.

6.3 Domain Classes

We design a set of domain classes used to describe the CTP model. The following are some main classes:

CTP.java an object of this class represents a CTP model.
Process.java an object of this class represents a process.
Scheme.java an object of this class represents a transactions scheme.
Transaction.java an object of this class represents a transaction
EventStr.java an object of this class represents an event structure
Event.java an object of this class represents an event in a MSC, or an event structure.

There are few more domain classes representing guards, variables, types, etc. All of the domain classes are put in ctp/ctpstr package.
6.4 Algorithmic Classes

We separate the domain classes (present elements of the CTP model structure) and algorithmic classes (present operations performed on the CTP model) to achieve the modularity and clarity in system architecture. Moreover, one of the most important objectives of system design is to allow define a new operation without changing the classes of elements which it operates - the domain classes. In this project, we use the Visitor pattern (E.Gamma et al., 1995) to do that. Next we present the use of Visitor pattern in our system.

With the Visitor pattern, whenever we want to add an operation to the system, for example, transaction scheme well-formed checking or transaction scheme to event structure transformation, we pack all the details of that operation in a separate object, called a visitor. After that we pass the visitor to the elements of the CTP model (Event, EventStr, Scheme, etc). When an element ”accepts” the visitor, it sends a request to the visitor’s method encoding the operation. It also includes the element itself as an argument, the visitor will then execute the operation for that element. It will be clearer by looking at the structure of class hierarchies in Figure 6.2.

![Figure 6.2. The Visitor pattern.](image-url)
Chapter 7
Discussions

In this thesis, we have presented our work on the CTP model, a high level executable specification language for modeling reactive systems, both theoretically and practically. The CTP model is based on Message Sequence Charts, which emphasize inter-process communication, and explicit description of intra-process. The CTP model is flexible, expressive and amenable to formal analysis and synthesis. The main goal of this project is to explore in depth the formal analysis/verification, and translation issues. The following sections will first summarize our contributions, then some future research directions.

7.1 Summary of the project

Our contributions to the model are as follows.

1. We have built a system to facilitate the use of the CTP model. Within current framework, a new operation on the CTP model can be easily added in the system.

2. We have constructed a translator that transforms a CTP into an intermediate representation (using Event Structures) which can be used in other modules, such as, transformation to SMV, hardware synthesis, etc.

3. We have also designed algorithms for checking the well-formedness of the result event structure. The notion of well-formedness is a useful criterion, by using which certain types of incomplete and inconsistent specifications at the level of transaction schemes can be caught.

4. Based on (Roychoudhury and Thiagarajan, 2002) we have provided the operational semantics of the CTP model in terms of coloured Petri nets. For each CTP we show how to construct an equivalent coloured Petri net which gives precisely the behaviour (operational semantics) of the system. The existence of an equivalent coloured Petri net is extremely useful. It tells us how to generalize the basic concepts and analysis methods of coloured Petri nets to CTP. We simply define
these concepts in such a way that a CTP has a given property if, and only if, the equivalent coloured Petri net has the corresponding property. Additionally, the existence of an equivalent coloured Petri net allows us to translate a CTP to a coloured Petri net, and use existing tools for verifying properties of the modeled system.

7.2 Future Work

There are many possibilities to continue our project. Here we list a few directions for future work in which we think they are worthy to investigate.

- Devise a more efficient and direct algorithm for determining the boundedness property. Even though, we are able to do so by translating CTP models to CPN and using existing tools. We believe that by making use of the structure of the CTP model we can devise a more efficient and direct method.

- Translate the CTP model to SMV – Symbolic Model Verifier – (SMV). The Petri nets representation of bounded CTP models can be represented as a finite transition system, and dynamic properties of the system can be specified in a temporal logic Roychoudhury and Thiagarajan (2002). Hence, we can carry out formal verification of these specifications using model checking tools such as SMV.

- Translate the CTP model to SPIN – another model checker – (SPIN). SPIN is a very powerful model checking tool too. This is an another interesting topic to pursue.

- Develop new features for the CTP model. The model is still in developing stage, there are many possibilities to extend the model, such as, adding object-oriented features, introducing the temporal logics to specify time constraints, etc.

- Develop schedulability analysis methods. Schedulability is an important aspect of real time systems, hence this issue definitely needs investigating.
REFERENCES


E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995. ISBN 0201633612.


