ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**D Duke, University of York, UK and A S Evans, University of Bradford, UK
(Eds)**

BCS-FACS Northern Formal Methods Workshop

Proceedings of the BCS-FACS Northern Formal Methods Workshop,
Ilkley, UK. 23-24 September 1996

Transformation of MTCCS into an Extension of Timed Automata

R.F. Lutje Spelberg and W.J. Toetenel

Springer

Published in collaboration with the
British Computer Society

Transformation of MTCCS into an Extension of Timed Automata

Ronald Lutje Spelberg and Hans Toetenel

Faculty of Technical Mathematics and Informatics, Delft University of Technology
Zuidplantsoen 4, 2628BZ Delft, The Netherlands

Abstract

This paper presents results from work in progress on finding a method for specification and formal verification of real-time concurrent systems incorporating a non-trivial data component. To model such systems, we use the real-time process algebra MTCCS, a Timed CCS variant, enhanced with a data manipulation model. Our verification method is based on the dual-language approach in which temporal logic is used to state properties over system models defined in some operational specification language. To obtain representations that allow the application of model checking, MTCCS specifications are transformed into XTGraphs, an extension of timed automata that allows the symbolic description of data. This paper discusses the specification language MTCCS and its translation to XTGraphs.

1 Introduction

In the verification of finite state systems, promising results have been reported in the application of model checking. Especially symbolic [6] and compositional techniques [4], have shown that verification of quite large systems is possible. These techniques are aimed at avoiding the state space explosion caused by interleaving of parallel processes. In recent years, real-time model checking techniques have been introduced [12, 26, 17]. These techniques focus on restraining the state space explosion originating from a system's timing aspects. While succesful in the verification of hardware systems, model checking has been less succesful in the verification of software systems. This is partly due to the fact that complex data components play a much more prominent role in software systems. Data components are the third major factor in the state space explosion problem, for which solutions have to be found. Our work is aimed at finding a verification technique that deals with all three aspects of state space complexity: interleaving, timing and data. The work presented here involves two aspects of such a technique. First the formal system specification language MTCCS is described. After that a transformation is described, which leads to a representation to which model checking techniques can be applied.

MTCCS [15, 18] (Model-oriented Timed Calculus of Communicating Systems), is a formal specification language aimed at defining real-time concurrent systems with a non-trivial data component. It is based on Timed CCS [31], a timed version of the process algebra notation CCS [21]. Timed CCS is combined with VDM-SL [14, 13], the specification language of VDM (Vienna Development Methodology), to allow manipulation of data components. MTCCS is based on earlier work on a language called MOSCA [27, 28, 29].

Our method for verifying property specifications will be based on model checking, which allows verification of temporal logic properties in a highly automatic manner. To apply model checking (especially symbolic model checking), typically a timed automaton-based system representation is needed. MTCCS specifications are therefore transformed into such a representation. This formalism, which we called XTGraphs (extended timed graphs), is a variant of timed automata [3], enhanced to symbolically represent data components. This two-phased approach to verification of process algebra specifications has been applied before (ATP [23], ET-LOTOS [8]), but not to formalisms that incorporate the formal definition of data components. The resulting XTGraph models are intended to serve as input for the subsequent symbolic model checking step. A secondary benefit of our transformation is that the resulting XTGraphs can be used as basis for other analysis techniques like simulation, or for the implementation of specifications.

Figure 1: Overview

Figure 1 presents an overview of how the work presented in this paper fits together. MTCCS specifications are mapped to a core syntax representation, on which the semantics of MTCCS is based. MTCCS specifications in core syntax form are transformed to XTGraphs. The correctness of our transformation is proved by constructing an equivalence relation between transition systems defined by the MTCCS semantics and transition systems defined by the XTGraph semantics. This relation is a strong equivalence [21], which is proved in [19].

The remainder of this paper is organized as follows. Section 2 discusses and defines our system specification formalism, section 3 presents the intermediate graph formalism, and section 4 gives the transformation. The final section describes how the work presented in this paper fits in a complete verification method, and briefly discusses complementary techniques that could be used to realize such a method.

It is assumed that the reader is more or less acquainted with VDM-SL. Throughout this paper VDM-SL is used to specify syntactic and semantic domains and semantic functions.

2 The MTCCS notation

This section gives an overview of the formal specification language MTCCS. MTCCS is the result of the merging of a process definition formalism (based on Timed CCS) and a data description formalism (VDM-SL).

2.1 Overview of the language

An MTCCS specification describes four aspects of systems of communicating processes: their data-containment, their functional behaviour, their process-structure and their behaviour in time. Associated with these aspects are the following MTCCS constructions: data type and state definitions, functions and operations on data, agent definitions, and timed actions. The former two are taken from VDM-SL, while the process and time aspects are based on Timed CCS.

The basic structuring element in the MTCCS model of a system is a process, called *agent*. The action sequence associated with an agent is called its *behaviour*. An overview of outer syntax constructions for specifying the behaviour of agents is shown in figure 2.

The first four constructions are prefix expressions, which are used to describe sequential behaviour. The *timed in prefix* and the *timed out prefix* are the timed prefixes from Timed CCS. These are CCS prefixes enhanced with a time variable that records the time between enabling and execution of the action.

Time identifiers of timed prefix constructions may occur as free variable within the behaviour expression following the $\odot$ operator. After taking the action of the prefix construction the actual value of the variable t is substituted in each free occurrence of the identifier within the behaviour expression following the $\odot$ operator. Thus in

$$act \star t \odot behaviour$$

Construction	Typical example
Timed in prefix	$\overline{action(variable)} \star var \odot \dots$
Timed out prefix	$\overline{action(value)} \star var \odot \dots$
Idle prefix	$idle\ value \odot \dots$
State manipulation prefix	$\sigma(statement) \odot \dots$
If expression	$if\ condition\ then\ \dots\ else\ \dots$
Guarded expression	$when\ condition\ do\ \dots$
Agent instantiation	$A\langle value \rangle$
Let expression	$let\ var = value\ in\ \dots$
Choice	$behaviour \oplus behaviour$
Parallel composition	$behaviour \mid behaviour$
Restriction	$behaviour \setminus \{labelset\}$
Relabeling	$behaviour[relabeling]$
Null agent	$null$

Figure 2: MTCCS outer syntax behaviour expressions

each occurrence of t in the behaviour expression $behaviour$ is bound by $\star t$. Upon taking the action act the actual value of the time variable t is substituted in all free occurrences of t in $behaviour$. Time progression is due to the taking of $idle$ actions, which is specified by means of the *idle prefix*. For example in

$$idle\ (delay) \odot \dots$$

specifies a prefix expression, in which taking the action causes the time to progress with the value $delay$. The domain for the values of the idle actions is $\mathbb{R}^{\geq 0}$, thus creating a dense time model.

The fourth type is the *state manipulation prefix*, which specifies a VDM-SL statement that manipulates the state of the agent in which the construction appears. For example

$$\sigma(stat) \odot R$$

instantaneously and atomically updates the state according to the statement $stat$ and subsequently behaves as specified by R .

The *If expression* allows conditional behaviour specification steered by data conditions. An expression

$$if\ x = 1\ then\ A\ else\ B$$

behaves as A if $x = 1$, and behaves as B otherwise. The evaluation of the expression is performed immediately.

The *guard construction* can be used to define processes that block on data conditions. The expression

$$when\ x = 1\ do\ A$$

delays until x becomes 1, and then immediately continues to behave as A .

An *Agent instantiation* expression defines the parameterized instantiation of behaviour expressions. A *Let expression* allows the definition of local value bindings, either fixed or loose. For example, in

$$let\ x = 3\ in\ A\langle x \rangle$$

the value 3 is bound to the identifier x in a deterministic way. The VDM component also allows *loose* expressions. In

$$let\ x = (\text{let } v \in \{1, 2, 3\} \text{ in } v) \text{ in } A\langle x \rangle$$

x is non-deterministically given a value. It may be either 1, 2, or 3. Looseness is also very useful to express delays that are not fixed but are defined as a value from some interval. For example

$$let\ d \in \{0 \dots 5\} \text{ in } idle(d) \odot \dots$$

```

agent Clock  $\langle \mathbb{R} \rangle$ 
ports in reset
      out time :  $\mathbb{N}$ 
state reading :  $\mathbb{N}$ 
local RClock  $\langle t \rangle \triangleq$ 
  (idle(t)  $\odot \sigma(\text{reading} := \text{reading} + 1) \odot \text{RClock } \langle t \rangle$ )
   $\oplus$ 
  (reset  $\odot \text{Clock } \langle t \rangle$ )
   $\oplus$ 
  ( $\overline{\text{time}}(\text{reading}) \star d \odot \text{idle}(t - d) \odot \sigma(\text{reading} := \text{reading} + 1) \odot \text{RClock } \langle t \rangle$ )
Clock  $\langle \text{tick} \rangle \triangleq \sigma(\text{reading} := 0) \odot \text{RClock } \langle \text{tick} \rangle$ 

```

Figure 3: Simple example Specification

defines a maximum delay of 5 time units.

The *choice*, *parallel composition*, *restriction*, and *relabeling* are standard CCS constructs. The first specifies non-deterministic choice between behaviours, while the other three are used to model parallelism and communication. Finally, the *Null agent* denotes the passive agent.

In MTCCS all actions except external actions (input or output actions without a counterpart in the specification) are urgent. An urgent action is performed immediately as soon as it is enabled, without letting time pass. Thus time may only pass if no actions can be performed. Notice furthermore that the *idle* construct plays a central role in the time model of MTCCS, as it is the only construct that drives the passing of time.

The VDM-SL component 'interfaces' with the process component in three ways: by value expressions, statements, and type definitions. The syntax domain *VE_{expr}* denotes all value expressions defined by the VDM-SL part of MTCCS. It offers basic types like natural numbers, integers, reals, booleans, product and union types, record types, optional types, function and operation types. Further it offers complex data structuring facilities based on set types, sequence types and map types. Value expressions can be found in for example let constructions and output actions. State manipulations are represented by the syntax domain *Statement*, and are used in the state manipulation prefix. Type definitions are used to define state definitions, port definitions, and value part definitions, which are all found in agent definitions.

At the top level, an MTCCS specification consists of one or more (global) agent definitions, and an initial behaviour expression. Global agents can only be instantiated in the initial behaviour expression. The initial behaviour expression can thus be seen as defining the main process structure. A global agent has a state and a number of local agents. The latter have no local state, but share the state of the global agent. They can only be instantiated from within the global agent they are defined in.

A global agent definition consists of a name, a port specification specifying type information for the actions of the agent, a state specification, a series of local agent definitions, and a behaviour expression defining its behaviour. Furthermore a value part is specified, which allows parametrised instantiation of the agent. A local agent is defined by a value part and a behaviour expression.

To express properties of MTCCS specifications we use a variant of TCTL (timed computation tree logic) [1]. The form of the property specification language is not relevant for the work presented here and therefore not further elaborated. A formal definition of it can be found in [18].

2.2 A small example

Figure 3 shows a small example MTCCS specification of a discrete clock. The first line states the agent heading, which is parameterized on the tick-length, typed through its value part $\langle \mathbb{R} \rangle$. The next two lines define two communication ports, the port *reset* and the port *time*. The agent *Clock* holds its current reading in the typed state variable *reading*.

The behaviour of the agent is specified through two agent behaviour definitions, *Clock* and *RClock*. *Clock* $\langle t \rangle$

defines the initial behaviour of the *Clock* agent. On instantiation the value part is bound to the name t . For example, the behaviour expression $Clock\langle 1 \rangle$ defines a clock that ticks with ratio 1, such that the duration of a tick is 1 time unit. The first action $\sigma(\text{reading} := 0)$ initializes the reading of the clock, after which the clock continues to behave like $RClock\langle r \rangle$. $RClock$ models one tick of the running clock. The behaviour of $RClock$ consists of three possible behaviours: it is merely ticking, or it may either partake in a synchronization through the *reset* action, or it may offer its reading in a synchronization through the *time* action. The first behaviour

$$(\text{idle}(t) \odot \sigma(\text{reading} := \text{reading} + 1) \odot RClock\langle t \rangle)$$

acts as time-out. If during the duration of a tick nothing happens then the idle action runs to completion, followed by the registration of a tick and the activation of a new tick cycle. The second variant

$$(\text{reset} \odot Clock\langle t \rangle)$$

restarts the clock. The third variant

$$(\overline{\text{time}}(\text{reading}) \star d \odot \text{idle}(t - d) \odot \sigma(\text{reading} := \text{reading} + 1) \odot RClock\langle t \rangle)$$

is activated on a synchronization on the *time* action. After performing the action, the remaining delay $(t - d)$ is taken.

2.3 Core abstract syntax

The core syntax of the process component of MTCCS is defined by the abstract domains given in figure 4. It forms the base syntax on which the MTCCS semantics are defined. The exact form of the data component (VDM-SL) is not relevant for the work presented here and is therefore omitted from this papers. The full definition of the core syntax of VDM-SL can be found in [13].

The core domain *Spec* defines a set of (global) agents, defined by *ProcDef* syntax constructs, and an initial behaviour. At abstract syntax level, there are two types of parallel compositions, one corresponding to a composition of global agents (*GPar*), and one corresponding to a composition of local agents (*LPar*). The first can only be used in the initial behaviour expression, while the latter is used in behaviour expressions in agents. Likewise, there are two types of agent instantiations (*GAgent* and *LAgent*).

Wait and *Delay* are core syntax domains without an outer syntax counterpart. They are intermediate forms, used in the semantics to give meaning to the passing of time.

2.4 Semantics

The semantics of MTCCS consists of two components; an operational semantics of the process part, and a denotational semantics of the data manipulation part. The latter is embodied by (a subset of) VDM-SL and is not discussed in this paper, but can be found in [13]. To be able to define the operational semantics of the process part, only the 'interface' between the two semantics needs to be defined.

A labeled transition system $M : (S, s_0, \Sigma, R)$ corresponding to an MTCCS specification consists of a set of states S , an initial state $s_0 \in S$, a set of labels Σ , and a transition relation R . A state is a tuple $\langle B, \rho, \nu \rangle \in (BExpr \times EnvT \times EnvT)$ consisting of three components: an MTCCS behaviour expression B , a local definitions valuation $\rho \in EnvT$, and a store valuation $\nu \in EnvT$. In ρ , local definitions originating from for example let expressions and input actions are defined. In ν , the actual valuation of the state component is defined.

$EnvT$ is defined as follows:

$$\begin{aligned} Env &= Id \xrightarrow{m} Val \\ EnvT &= EnvT \cdot EnvT \mid Env \end{aligned}$$

$EnvT$ represents trees of environments, which are patterned after the process structure. As can be seen from the semantics rules given below, every parallel composition causes the local environment to split, creating a local environment for each subprocess. A choice may also cause the local environment to split, but this is only temporary. The

$Spec :: \text{procs} : \text{ProcId} \xrightarrow{m} \text{ProcDef}$ $\text{behaviour} : \text{BExpr}$		
$ProcDef :: \text{state} : \text{VarId} \xrightarrow{m} (\text{Type} \times \text{VExpr})$ $\text{locals} : \text{ProcId} \xrightarrow{m} (\text{BExpr} \times \text{Type} \times \text{VarId})$ $\text{behaviour} : \text{BExpr} \times \text{Type} \times \text{VarId}$		
$BExpr = \text{InAct} \mid \text{OutAct} \mid \text{Idle} \mid \text{StateMan} \mid \text{Let} \mid \text{LAgent} \mid \text{GAgent} \mid \text{Guard} \mid \text{If} \mid$ $\text{Choice} \mid \text{GPar} \mid \text{LPar} \mid \text{Restrict} \mid \text{Relab} \mid \text{Null} \mid \text{Wait} \mid \text{Delay}$		
$\text{InAct} :: \text{label} : \text{ActId}$ $\text{mvar} : \text{VarId}$ $\text{dvar} : \text{VarId}$ $\text{cont} : \text{BExpr}$	$\text{OutAct} :: \text{label} : \text{ActId}$ $\text{mvar} : \text{VarId}$ $\text{dvar} : \text{VarId}$ $\text{cont} : \text{BExpr}$	$\text{Stateman} :: \text{sm} : \text{Statement}$ $\text{cont} : \text{BExpr}$
$\text{Let} :: \text{var} : \text{VarId}$ $\text{val} : \text{VExpr}$ $\text{cont} : \text{BExpr}$	$\text{LAgent} :: \text{name} : \text{ProcId}$ $\text{val} : [\text{VExpr}]$	$\text{GAgent} :: \text{name} : \text{ProcId}$ $\text{val} : [\text{VExpr}]$
$\text{Idle} :: \text{durval} : \text{VExpr}$ $\text{cont} : \text{BExpr}$	$\text{Guard} :: \text{guard} : \text{VExpr}$ $\text{action} : \text{BExpr}$	$\text{If} :: \text{cond} : \text{VExpr}$ $\text{cont} : \text{BExpr}$ $\text{alt} : \text{BExpr}$
$\text{Choice} :: \text{left} : \text{BExpr}$ $\text{right} : \text{BExpr}$	$\text{GPar} :: \text{left} : \text{BExpr}$ $\text{right} : \text{BExpr}$	$\text{LPar} :: \text{left} : \text{BExpr}$ $\text{right} : \text{BExpr}$
$\text{Restrict} :: \text{res} : \text{ActId-set}$ $\text{cont} : \text{BExpr}$	$\text{Relabeling} :: \text{rel} : \text{ActId} \xrightarrow{m} \text{ActId}$ $\text{cont} : \text{BExpr}$	
$\text{Wait} :: \text{timer} : \mathbb{R}^{\geq 0}$ $\text{cont} : \text{BExpr}$	$\text{Delay} :: \text{act} : \text{InAct} \mid \text{OutAct}$ $\text{delay} : \mathbb{R}^{\geq 0}$	

Figure 4: Syntax of MTCCS

reason for this is that some actions do not resolve the choice, but do already change the local environments of the components of the choice.

The store environment is only split in case of a global parallel composition ($GPar$), since local agents share the state environment of the global agent they are defined in.

The interface between the operational semantics and the denotational semantics consists of two components; the evaluation of state manipulations, and the evaluation of value expressions. This interface is realized by the semantical function $eval$ which is defined as follows:

$$eval : \begin{cases} \text{Statement} & \rightarrow (\text{Env} \times \text{Env} \rightarrow \text{Env})\text{-set} \\ \text{VExpr} & \rightarrow (\text{Env} \times \text{Env} \rightarrow \text{Val})\text{-set} \end{cases}$$

In the case of a state manipulation, the semantical function takes a *Statement* and returns a set of (semantical) state transformers. A state transformer takes a local environment and a store environment and returns a new store environment. In the case a value expression (*VExpr*) is evaluated a set of evaluators is returned. Each evaluator takes a local and a store environment and returns a value.

A transition relation $R \subseteq S \times \Sigma \times S$ is a set of transitions (s, l, s') , where $l \in \Sigma$ is a transition label. The set of transition labels Σ is defined as follows:

- $!val, l?val \in (\text{ActId} \times \text{Val})$: external actions

SPEC	$Spec(bh, st, init) \xrightarrow{t} \langle init, \emptyset, \emptyset \rangle$
STATEMAN	$[\mu \in eval[[sm]]] \langle StateMan(sm, B), \rho, \nu \rangle \xrightarrow{\gamma} \langle B, \rho, \mu(\rho, \nu) \rangle$
IDLE1	$[d = ev(\rho, \nu), ev \in eval[[e]]] \langle Idle(e, B), \rho, \nu \rangle \xrightarrow{t} \langle Wait(d, B), \rho, \nu \rangle$
IDLE2	$\langle Wait(d, B), \rho, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle Wait(d-\delta, B), \rho, \nu \rangle, d > \delta$
IDLE3	$\langle Wait(d, B), \rho, \nu \rangle \xrightarrow{\epsilon(d)} \langle B, \rho, \nu \rangle$
ACTDELAY1	$\langle act, \rho, \nu \rangle \xrightarrow{t} \langle Delay(act, 0), \rho, \nu \rangle \quad act \in \{InAct(-, -, -, -), OutAct(-, -, -, -)\}$
ACTDELAY2	$\langle Delay(act, d), \rho, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle Delay(act, d + \delta), \rho, \nu \rangle \quad act \in \{InAct(-, -, -, -), OutAct(-, -, -, -)\}$
INACT	$\langle Delay(InAct(l, v, t, B), d), \rho, \nu \rangle \xrightarrow{tval} \langle B, \rho[v \mapsto val, t \mapsto d], \nu \rangle$
OUTACT	$[ev \in eval[[e]], val = ev(\rho, \nu)] \langle Delay(OutAct(l, e, t, B), d), \rho, \nu \rangle \xrightarrow{tval} \langle B, \rho[t \mapsto d], \nu \rangle$
LET	$[\rho' = \rho[v \mapsto ev(\rho, \nu)], ev \in eval[[e]]] \langle Let(v, e, B), \rho, \nu \rangle \xrightarrow{t} \langle B, \rho', \nu \rangle$
LAGENT	$[Behav(P) = (B, v), \rho' = \{v \mapsto ev(\rho, \nu)\}, ev \in eval[[e]]] \langle LAgent(e, P), \rho, \nu \rangle \xrightarrow{t} \langle B, \rho', \nu \rangle$
GAGENT	$\left[\begin{array}{l} Behav(P) = (B, v), \nu' \in Default(P), \\ \rho' = \{v \mapsto ev(\rho, \emptyset)\}, ev \in eval[[e]] \end{array} \right] \langle GAgent(e, P), \rho, \nu \rangle \xrightarrow{t} \langle B, \rho', \nu' \rangle$
GUARD1	$[ev \in eval[[cond]], ev(\rho, \nu) = true] \frac{\langle B, \rho, \nu \rangle \xrightarrow{x} \langle B', \rho', \nu' \rangle, x \neq \epsilon(-)}{\langle [g]B, \rho, \nu \rangle \xrightarrow{x} \langle B', \rho', \nu' \rangle}$
GUARD2	$[ev \in eval[[cond]], ev(\rho, \nu) = false] \langle [g]B, \rho, \nu \rangle \xrightarrow{\epsilon(-)} \langle [g]B, \rho, \nu \rangle$
GUARD3	$\frac{\langle B, \rho, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B', \rho, \nu \rangle}{\langle [g]B, \rho, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle [g]B', \rho, \nu \rangle}$
IF1	$[ev \in eval[[cond]], ev(\rho, \nu) = true] \langle If(cond, B_1, B_2), \rho, \nu \rangle \xrightarrow{t} \langle B_1, \rho, \nu \rangle$
IF2	$[ev \in eval[[cond]], ev(\rho, \nu) = false] \langle If(cond, B_1, B_2), \rho, \nu \rangle \xrightarrow{t} \langle B_2, \rho, \nu \rangle$
CHOICE1	$\langle Choice(B_1, B_2), \rho, \nu \rangle \xrightarrow{t} \langle Choice(B_1, B_2), \rho \cdot \rho, \nu \rangle$
CHOICE2	$\frac{\langle B_i, \rho, \nu \rangle \xrightarrow{x} \langle B'_i, \rho', \nu' \rangle, x \notin \{\epsilon(-), \iota\}}{\langle Choice(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{x} \langle B'_i, \rho'_i, \nu' \rangle} \quad i \in \{1, 2\}$
CHOICE3	$\frac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B'_1, \rho_1, \nu \rangle, \langle B_2, \rho_2, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B'_2, \rho_2, \nu \rangle}{\langle Choice(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle Choice(B'_1, B'_2), \rho'_1 \cdot \rho'_2, \nu \rangle}$
CHOICE4	$\frac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{t} \langle B'_1, \rho'_1, \nu \rangle}{\langle Choice(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{t} \langle Choice(B'_1, B_2), \rho'_1 \cdot \rho_2, \nu \rangle} \quad 1$

Figure 5: Semantics of MTCCS (1)

- γ : Visible internal actions (as a consequence of internal communications and state manipulations)
- ι : Invisible internal actions (as a consequence of evaluation of expressions in *Idle*, *AgentIf* and *Let* constructions, and expansion of environments in *Choice* and *ParComp* expressions. These actions do not resolve Choices.
- $\epsilon(\delta), \delta \in \mathbb{R}^{\geq 0}$: Time actions

For clarity we use only single values in input actions, output actions, let constructions, and value parts. The extension to more complex expressions is straightforward and does not have any consequences for the discussions presented here.

The semantics is given by structural operational semantics interference rules. The transition relation of the transition system corresponding to an MTCCS specification *Spec* is defined as the least relation satisfying the rules given in figures 5, and 6.

The first rule (SPEC) constructs the initial state of *Spec* defined by the initial behaviour expression *Spec.init*, an empty initial Store valuation, and an empty local definitions valuation. An additional environment that holds the

²The dual rule is not shown

GPAR1	$\langle GPar(B_1, B_2), \rho, \nu \rangle \xrightarrow{\iota} \langle GPar(B_1, B_2), \rho \cdot \rho, \nu \cdot \nu \rangle$
GPAR2	$\frac{\langle B_1, \rho_1, \nu_1 \rangle \xrightarrow{x} \langle B'_1, \rho'_1, \nu'_1 \rangle, x \neq \epsilon(-)}{\langle GPar(B_1, B_2), \rho_1 \cdot \rho_2, \nu_1 \cdot \nu_2 \rangle \xrightarrow{x} \langle GPar(B'_1, B'_2), \rho'_1 \cdot \rho'_2, \nu'_1 \cdot \nu'_2 \rangle}^2$
GPAR3	$\frac{\langle B_1, \rho_1, \nu_1 \rangle \xrightarrow{\iota^? val} \langle B'_1, \rho'_1, \nu'_1 \rangle, \langle B_2, \rho_2, \nu_2 \rangle \xrightarrow{\iota^? val} \langle B'_2, \rho'_2, \nu'_2 \rangle}{\langle GPar(B_1, B_2), \rho_1 \cdot \rho_2, \nu_1 \cdot \nu_2 \rangle \xrightarrow{\gamma} \langle GPar(B'_1, B'_2), \rho'_1 \cdot \rho'_2, \nu'_1 \cdot \nu'_2 \rangle}$
GPAR4	$\frac{\langle B_1, \rho_1, \nu_1 \rangle \xrightarrow{\epsilon(\delta)} \langle B'_1, \rho_1, \nu_1 \rangle, \langle B_2, \rho_2, \nu_2 \rangle \xrightarrow{\epsilon(\delta)} \langle B'_2, \rho_2, \nu_2 \rangle, \langle GPar(B_1, B_2), \rho_1 \cdot \rho_2, \nu_1 \cdot \nu_2 \rangle \xrightarrow{\gamma, \iota}}{\langle GPar(B_1, B_2), \rho_1 \cdot \rho_2, \nu_1 \cdot \nu_2 \rangle \xrightarrow{\epsilon(\delta)} \langle GPar(B'_1, B'_2), \rho_1 \cdot \rho_2, \nu_1 \cdot \nu_2 \rangle}$
LPAR1	$\langle LPar(B_1, B_2), \rho, \nu \rangle \xrightarrow{\iota} \langle LPar(B_1, B_2), \rho \cdot \rho, \nu \rangle$
LPAR2	$\frac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{x} \langle B'_1, \rho'_1, \nu' \rangle, x \neq \epsilon(-)}{\langle LPar(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{x} \langle LPar(B'_1, B'_2), \rho'_1 \cdot \rho'_2, \nu' \rangle}^2$
LPAR3	$\frac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{\iota^? val} \langle B'_1, \rho'_1, \nu' \rangle, \langle B_2, \rho_2, \nu \rangle \xrightarrow{\iota^? val} \langle B'_2, \rho'_2, \nu' \rangle}{\langle LPar(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\gamma} \langle LPar(B'_1, B'_2), \rho'_1 \cdot \rho'_2, \nu' \rangle}$
LPAR4	$\frac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B'_1, \rho_1, \nu \rangle, \langle B_2, \rho_2, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B'_2, \rho_2, \nu \rangle, \langle LPar(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\gamma, \iota}}{\langle LPar(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle LPar(B'_1, B'_2), \rho_1 \cdot \rho_2, \nu \rangle}$
RESTRICT	$\frac{\langle B, \rho, \nu \rangle \xrightarrow{x} \langle B', \rho', \nu' \rangle, x \notin res}{\langle Restrict(res, B), \rho, \mu \rangle \xrightarrow{x} \langle Restrict(res, B'), \rho', \nu' \rangle}$
RELAB	$\frac{\langle B, \rho, \nu \rangle \xrightarrow{x} \langle B', \rho', \nu' \rangle}{\langle Relab(rel, B), \rho, \nu \rangle \xrightarrow{x[rel]} \langle B', \rho', \nu' \rangle}$
NULL	$\langle Null, \rho, \nu \rangle \xrightarrow{\epsilon(d)} \langle Null, \rho, \nu \rangle$

Figure 6: Semantics of MTCCS (2)

bindings of process identifiers to behaviour expressions is left implicit in the semantic rules. This environment is assumed to be created by the SPEC rule and is used in the LAGENT, and GAGENT rules. In the latter the existence of a function $Behav : ProcId \rightarrow (BExpr \times VarId)$ is assumed which given an agent name, returns the associated behaviour expression and value part variable. Furthermore a function $Default : ProcId \rightarrow Env$ is assumed which returns the initial store valuation corresponding to a global agent.

The STATEMAN rule defines the behaviour associated with the state manipulation prefix. The state manipulation statement sm is evaluated through the evaluation function $eval$, resulting in a set state transformers. For each element $\mu \in eval[s_m]$ there is a possible action

$$\langle StateMan(sm, B), \rho, \nu \rangle \xrightarrow{\gamma} \langle B, \rho, \mu(\rho, \nu) \rangle$$

After performing a γ action a transition system state is reached defined by the the continuation behaviour B , an unchanged ρ , and a Store ν updated according to μ .

The next three rules define the semantics of the idle prefix. The first rule (IDLE1) defines the action that captures the evaluation of the value expression denoting the delay. The evaluation of the expression results in a set of evaluators. Applying an evaluator to ρ and ν results in a delay value d . The action results in a state containing the abstract syntax construction $Wait$ which holds the semantic value of the delay (d) and the continuation behaviour. IDLE2 describes the effect of delaying: with every delay action $\epsilon(\delta)$ the delay value of the $Wait$ construct is decreased with δ . IDLE3 defines the final delay action that completes the idling corresponding to the idle construct. The separate ι action corresponding to the evaluation of the delay expression (IDLE1) is needed because the value of the expression may change due to other processes changing the global state.

The semantics of the external action prefixes are defined by the next four rules. ACTDELAY1 defines the transformation of an action construct ($InAct$ or $OutAct$) into a delay construct which contains the original action construct and a value corresponding to the accumulated idle time. The latter is initialized to zero. ACTDELAY2 rule defines the actual delaying. A delay action causes the delay value to be increased with the amount time that is delayed. The

²The dual rule is not shown

other two rules correspond to the actual execution of the input or output actions. The execution of an input action (INACT) results in an updated local definitions environment. The variable v is bound to the value corresponding to the input action (val), and the variable t is bound to d , the accumulated delay value. When performing an output action (OUTACT), the accumulated delay value is also bound to the time variable. Furthermore, the output value (val) is obtained by evaluating the expression corresponding to the output action (e).

The rule for the *let* construction is straightforward. A ι action is performed resulting in an updated local definitions environment ρ' in which the binding defined by the *let* expression is captured.

The rule corresponding to the *local agent instantiation* (LAGENT) uses the earlier mentioned *Behav* function to obtain the agent instantiation corresponding to a *ProcId*. A new local definitions environment is created containing only the binding that is the result of the value part of the agent instantiation. The store environment is left unchanged. The *global agent instantiation* rule (GAGENT) is similar to that of local agents, but also initializes the store environment.

The semantics rules of the guard (GUARD1-3) capture the fact that a guard whose condition is satisfied, is removed together with the first (non-delay) action of the subsequent behaviour expression. This is what the first rule effects. The second rule defines the passing of time in case the condition is not satisfied, and the third rule defines that a guarded expression may pass time if the subsequent behaviour expression can.

The semantics of the *if* expression is captured by rules IF1 and IF2. An *if* expression performs a ι action resulting in the continuation behaviour that corresponds to the value of the condition. Again the evaluation of the expression is modeled by a separate ι action, because the store can be updated by other processes, causing the value of the condition to change.

The semantics of the *choice* expression is defined by four rules. CHOICE1 effects a splitting of the environment, one for each side of the choice. This is needed because not all actions resolve the choice, but may add local definitions (for example the ι action of an idle construct). The splitting is resolved as soon as an external action or a γ action is performed on either side of the choice. The second rule describes the effect of these actions. If a branch can perform such an action, the whole choice can perform that action, resolving the choice. CHOICE3 describes the idling of a choice construct, while the fourth rule defines the effect of a ι action, which does not resolve the choice.

Parallel composition is described by means of four rules. There is a set of rules for the global parallel composition and one for the local parallel composition. The only difference is that the local variant only splits the local definitions environment, while the global variant also splits the store environment. The first rules initialise these splittings: LPAR1 splits ρ in to equivalent environments, and GPAR1 also splits ν . LPAR2 and GPAR2 describe the effect of performing actions that are not communications or delays, LPAR3 and GPAR3 capture communication, and LPAR4 and GPAR4 define how time may pass. From the latter two rules it can be seen that a parallel composition can only idle if no γ or ι actions are enabled, which is expressed by $\langle Par(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\gamma, \iota}$. This enforces the *urgency* of γ and ι actions; if such an action is enabled, it is performed immediately. The use of a negative premises may in general lead to inconsistencies in the resulting transition system [10]. Our use of a negative premise in LPAR4 and GPAR4 can easily be proved correct following the approach given in [10].

RESTRICT and RELAB are the standard CCS rules for restriction and relabeling. $x[rel]$ denotes the application of the relabeling rel to action x . The *Null* agent can only idle, which is expressed by NULL.

The semantics given here is not complete as it does account for the fact that the evaluation of value expressions may result in erroneous values. Therefore also a number of *divergence* rules are defined which describe the behaviour in case erroneous values or type inconsistencies are encountered. The complete set of semantics rules is given in [18].

3 XTGraphs

XTGraphs can be viewed as an extension of timed automata [1]. A timed automaton is a finite state transition graph equipped with a finite set of continuous clocks which all run at the same rate. Transitions may reset clocks and are guarded by means of clock constraints referring to current values of the clocks. XTGraphs are timed automata extended with capabilities to operate on data, by adding data manipulations and data constraints to transitions.

Many variants of timed automata exist. Differences between these approaches often stem from the way the passing of time is constrained. Timed Safety Automata [12] have constraints associated with locations. For example a

constraint $X \leq 3$ forces the system to move from that location before the constraint becomes false. Timed Graphs [23] have a divergence condition $div \in \{true, false\}$ associated with each location. A divergence condition $true$ indicates that the system may always remain in the corresponding location. If the divergence condition is $false$, then time may only pass if this does not lead to a state in which no more actions are possible.

We use yet another approach in which time may only pass, if no actions are enabled. This means that every enabled action is performed immediately, without letting time pass. This results in a simple time model that fits well to models defined by closed MTCCS specifications, since in a closed system all actions are internal and therefore urgent. Urgent actions are performed immediately upon enabling. This restriction is not fundamental, since one could easily extend the XTGraph model to also allow external actions. As a consequence, time constraints are always lower bounds on clock values.

When enhancing timed automata with data, choices have to be made on how to incorporate the data component. Our approach to adding a data model is analogous to the approach used for MTCCS. We again assume a denotational semantics interface with the data model. Other approaches usually embed the data model in the operational semantics (for example [11, 16]). An advantage of our model is that it does not make many assumptions on the form of the data model, only its abstract interface is defined. This allows us to use the existing data model of VDM. Important for us is that our model fits very closely to the data model of MTCCS. This allows us to concentrate on the operational part of our models, abstracting to some extent from the data manipulation part.

First a general version of an XTGraph is defined for which little assumptions are made on how data is modeled. After that, a specific interpretation focused on our type of system models is given.

Manipulation of data is modeled by adding environment updates to edges. Furthermore, data-dependent behaviour is modeled by associating data constraints with edges. An XTGraph is a tuple $\langle L, l_0, X, T, E \rangle$, where

- L is a set of locations, with initial location l_0
- X is a set of clocks
- $T : L \rightarrow \underline{S}$ assigns a type signature to every location
- E is a set of edges: $E \subseteq L \times L \times A \times U \times C$

where

- A is a set of action labels
- $\underline{S}$ is the set of tuples of sorts. S is the set of sorts.
- C is the set of constraints (on clocks and data).
- U is the set of state updates (clocks resets and data manipulations)

States of an XTGraph transition system are tuples $\langle l, v \rangle \in L \times \text{Envir}$, where l is the location, and v is an environment defined as follows:

$$\begin{aligned} \text{Envir} :: \text{clocks} : Id &\xrightarrow{m} \mathbb{R}^{\geq 0} \\ \text{data} : Id &\xrightarrow{m} \text{Val} \end{aligned}$$

At this stage, we do not commit to a data model, but only define the format of the interface. This interface is embodied by two denotational semantic functions: one for the evaluation of state and clock updates and one for the evaluation of constraints:

$$\begin{aligned} \text{EvalU} : U &\rightarrow (\text{Envir} \rightarrow \text{Envir})\text{-set} \\ \text{EvalC} : C &\rightarrow (\text{Envir} \rightarrow \mathbb{B}) \end{aligned}$$

Note that loose state updates are possible: the evaluation of an update returns a set of state transformers.

We can now define the semantics of XTGraphs. In the following, $\rho[+\delta]$ is a function $(\text{Envir} \times \mathbb{R}^{\geq 0}) \rightarrow \text{Envir}$, that increases each clock in $\rho.\text{clocks}$ with δ .

The transition relation of an XTGraph $G = (L, l_0, X, T, E)$ is the least relation defined by the following rules:

$$\frac{(l, l', a, u, c) \in E, \mu \in EvalU(u), EvalC(c)v = true}{\langle l, v \rangle \xrightarrow{a} \langle l', \mu(v) \rangle} \quad \frac{\forall 0 < d \leq \delta \cdot \langle l, v[+d] \rangle \not\xrightarrow{a}, a \neq \epsilon(-)}{\langle l, v \rangle \xrightarrow{\epsilon(\delta)} \langle l, v[+\delta] \rangle}$$

The first rule states that a transition corresponding to an edge is executed if its constraint is satisfied. As a consequence, the state is updated according to the state update belonging to the edge. According to the second rule, time may pass in a location as long as no actions are enabled.

3.1 A tailored interpretation of XTGraphs

Because the graphs resulting from our expansions have some specific characteristics, we use a interpretation of the XTGraphs defined above, which is less general and more tailored towards the kind of models that result from expansion of MTCCS specifications.

The data component of the environment consists of two parts: a store environment and a local definitions environment. The store environment is a single mapping from variables to values, whereas the local definitions environment is explicitly split into components according to the process structure of the system. Unlike an MTCCS specification does an XTGraph describe a global model of a system. As a consequence, the selection of proper local definition environments has to be made explicit. Therefore local definition environments are identified by 'process identifiers' (*PId*'s). Local parallel compositions in the specifications cause the local definitions environment to change dynamically. The store environment can be modeled as a single valuation, because its form is determined in the initial behaviour expression.

When expanding an MTCCS specification, exactly one clock will be needed for each process, since within a single sequential MTCCS process, all timing constraints are local, that is, defined between two successive actions. For this reason we simply use process identifiers to identify clocks. The resulting environment then looks as follows:

$$\begin{aligned} Envir :: Clocks : PId &\xrightarrow{m} \mathbb{R}^{\geq 0} \\ Store : VarId &\xrightarrow{m} Val \\ Locals : PId &\xrightarrow{m} (VarId \xrightarrow{m} Val) \end{aligned}$$

We can also define specific forms for updates (*U*) and constraints (*C*). Figure 7 presents the definitions of environment updates and constraints. An update $u \in U$ is a set of zero or more of the above defined single updates. A

$U = Upd\text{-set}$	$Upd = SM \mid Loc \mid Clk \mid New \mid Res \mid Split$		
$SM :: pid : PId$ $sm : Statement$	$Loc :: pid : PId$ $var : VarId$ $val : VExpr$	$Clk :: pid : PId$ $var : VarId$ $del : VarId$	
$New :: pid : PId$	$Res :: pid : PId$	$Split :: pid, pid' : PId$	
$C = Con\text{-set}$	$Con = CCon \mid DCon$		
$CCon :: pid : PId$ $var : VarId$	$DCon :: pid : PId$ $con : Vexpr$		

Figure 7: Updates and constraints

single update (*Upd*) associated with MTCCS specifications is always one of five forms:

- *SM*: Changing the *Store* component of the environment. An *SM* update corresponds to a state manipulation. It consists of a statement and a process identifier which indicates in which local environment the statement is to be evaluated.

- *Loc*: Adding a local definition to a local definitions environment (as a consequence of a let construction or the evaluation of the expression in an idle construct. A *Loc* update consists of a variable name, a value expression and a process identifier.
- *Loc**: Similar to *Loc*, the only difference being that the old local definitions environment can be disposed of (as a consequence of an agent instantiation).
- *Clk*: Binding the value of a clock to a local definition identifier. A *Clk* update consists of a *PI* identifying a clock, and two variable identifiers *var* and *del*. It corresponds to the update $var := C_{pid} + del$, where C_{pid} is the clock identified by *pid*.
- *Res*: Resetting a clock.
- *Split*: splitting a local definitions environment (as a consequence of parallel compositions and choice constructs). A *Split* update consist of two process identifiers, the first corresponding to an existing local definitions environment, and the second corresponding to the new copy.

A constraint $c \in C$ is a set of clock constraints (*CCon*) and data constraints (*DCon*). A clock constraint (pid, var) represents an inequality of the form $clock \geq variable$ (remember that pids identify clocks). These constraints originate from *idle* expressions. A data constraint (pid, con) is a VDM-SL value expression (*con*), together with a pid indicating in which local definitions environment the constraint is evaluated. A data constraint originates from an *if* expression.

Finally, the evaluation functions *EvalU* and *EvalC* have to be defined. They are fairly straightforward. Appendix A presents VDM-SL definitions of these functions.

4 The transformation

This chapter discusses the transformation of MTCCS specifications to XTGraphs. Similar transformations have been defined for real-time process algebras without data (ATP [22, 23], a subset of ET-LOTOS [8]). Schreiber [25] describes a transformation from value-passing CCS to parameterized graphs, a graph representation equipped with symbolic data values. In [5] a similar transformation is described for the value passing process algebra μ CRL. Both latter approaches deal with non-real-time systems.

The expansion is defined by a set of interference rules. Given an MTCCS specification, the set of edges of the corresponding XTGraph is defined by the smallest set of edges satisfying the interference rules. In the rules, edges are identified by what we will call symbolic transitions: $B \xrightarrow{(l,u,c)}_S B'$ denotes an edge between locations identified by *B* and *B'*, with label *l*, update *u*, and constraint *c*. Locations are identified by syntactic constructs that are derived from the MTCCS core abstract syntax constructs. For this we use the abstract constructs that are part of the outer syntax (thus all those defined in figure 4 except the *Wait* and *Delay* construct) and a slightly different *Wait* construct, and a *Par* construct:

$$\begin{array}{ll} \text{Wait} :: \text{tvar} : \text{VarId} & \text{Par} :: \text{left} : \text{BExpr} \\ & \text{right} : \text{Bexpr} \\ & \text{cont} : \text{BExpr} \end{array}$$

These constructs may be superscripted with a process identifier, which is done to make sure that the correct local definitions environment is selected.

Analogous to the semantics rules of figures 5 and 6, we use *B*, *B'*, *B₁*, and *B₂* to range over the constructs. Furthermore, with B^β we denote the construct identified by *B* superscripted by β . Thus if $B = \text{Idle}(2, B')$ then B^β denotes $\text{Idle}^\beta(2, B')$.

We assume that identifiers can always be recognized as either belonging to the state or as being used for a local definition. This can be ensured by the mapping from the outer syntax to the core syntax.

To ensure that the translation terminates, the (control space of the) system has to be finite state. For this reason we do not allow recursion through the parallel composition operator, the restriction operator, and the relabeling operator, which is a commonly applied restriction.

ID1*	$Idle^\beta(e, B) \xrightarrow{(\iota, \{Loc(\beta, v, e)\}, \emptyset)} Wait^\beta(v, B)$ ³	ID2*	$\frac{B^\beta \xrightarrow{(x, u, c)} B'^\beta, x \in \{\iota, \gamma\}}{Wait^\beta(v, B) \xrightarrow{(x, u \cup \{Res(\beta)\}, c \cup \{CCon(\beta, v)\})} B'^\beta}$
ID3*	$\frac{B^\beta \xrightarrow{(l?(\beta, w, t, 0), \emptyset, c)} B'^\beta}{Wait^\beta(v, B) \xrightarrow{(l?(\beta, w, t, v), \emptyset, c \cup \{CCon(\beta, v)\})} B'^\beta}$		
IN*	$InAct^\beta(l, v, t, B) \xrightarrow{(l?(\beta, v, t, 0), \emptyset, \emptyset)} B^\beta$	OUT*	$OutAct^\beta(l, e, t, B) \xrightarrow{(l!(\beta, e, t, 0), \emptyset, \emptyset)} B^\beta$
LET*	$LetExpr^\beta(v, e, B) \xrightarrow{(\iota, \{Loc(\beta, v, e)\}, \emptyset)} B^\beta$	SM*	$StateMan^\beta(sm, B) \xrightarrow{(\gamma, \{SM(\beta, sm)\}, \emptyset)} B^\beta$
GAG*	$\frac{Behav(P) = (B, v)}{GAgent(P, e) \xrightarrow{(\iota, \{Loc^*(\beta, v, e)\}, \emptyset)} B^\beta}$ ⁴	LAG*	$\frac{Behav(P) = (B, v)}{LAgent^\beta(P, e) \xrightarrow{(\iota, \{Loc^*(\beta, v, e)\}, \emptyset)} B^\beta}$
GRD*	$\frac{B^\beta \xrightarrow{(x, u, c)} B'^\beta}{Guard^\beta(g, B) \xrightarrow{(x, u \cup \{Res(\beta)\}, c \cup \{Dcon(\beta, g)\})} B'^\beta}$		
IF1*	$If^\beta(c, B_1, B_2) \xrightarrow{(\iota, \emptyset, \{DCon(Not(c))\})} B_2^\beta$	IF2*	$If^\beta(c, B_1, B_2) \xrightarrow{(\iota, \emptyset, \{DCon(c)\})} B_1^\beta$
CH1*	$Choice^\beta(B_1, B_2) \xrightarrow{(\iota, \{Split(\beta, \beta'), Res(\beta')\}, \emptyset)} Choice(B^\beta, B'^\beta)$		
CH2*	$\frac{B_1 \xrightarrow{(x, u, c)} B_1'}{Choice(B_1, B_2) \xrightarrow{(x, u, c)} Choice(B_1', B_2)}$ ⁵	CH3*	$\frac{B_i \xrightarrow{(x, u, c)} B_i', x \neq \iota}{Choice(B_1, B_2) \xrightarrow{(x, u, c)} B_i} i \in \{1, 2\}$
PAR1G*	$GPar(B_1, B_2) \xrightarrow{(\iota, \emptyset, \emptyset)} Par(B_1, B_2)$		
PAR1L*	$LPar^\beta(B_1, B_2) \xrightarrow{(\iota, \{Split(\beta, \beta'), Res(\beta')\}, \emptyset)} Par(B_1^\beta, B_2^\beta)$ ⁴		
PAR2*	$\frac{B_1 \xrightarrow{\gamma} B_1'}{Par(B_1, B_2) \xrightarrow{\gamma} Par(B_1', B_2)}$ ⁵		
PAR3*	$\frac{B_1 \xrightarrow{(l?(\beta_1, v, t_1, d_1), \cdot, c_1)} B_1', B_2 \xrightarrow{(l!(\beta_2, e, t_2, d_2), \cdot, c_2)} B_2'}{Par(B_1, B_2) \xrightarrow{(\gamma, \{Res(\beta_1), Res(\beta_2), Loc(\beta_1, v, e), Clk(\beta_1, t_1, d_1), Clk(\beta_2, t_2, d_2)\}, c_1 \cup c_2)} Par(B_1', B_2')}$		
RS1*	$\frac{B \xrightarrow{(l?(\beta, v, t, d), u, c)} B', l \notin rs}{Restr(rs, B) \xrightarrow{(l?(\beta, v, t, d), u, c)} Restr(rs, B')}$ ⁶	RS2*	$\frac{B \xrightarrow{(x, u, c)} B', x \in \{\iota, \gamma\}}{Restr(rs, B) \xrightarrow{(x, u, c)} Restr(rs, B')}$
RL1*	$\frac{B \xrightarrow{(l?(\beta, v, t, d), u, c)} B'}{Relab(rl, B) \xrightarrow{(l[rl]?(\beta, v, t, d), u, c)} Relab(rl, B')}$ ⁶	RL2*	$\frac{B \xrightarrow{(x, u, c)} B', x \in \{\iota, \gamma\}}{Relab(rl, B) \xrightarrow{(x, u, c)} Relab(rl, B')}$

Figure 8: Transformation rules

As mentioned before, we define the transformation for closed MTCCS specifications. This results in a simpler XTGraph model and does not impose any severe restrictions, since the non-deterministic environment that comes with an open system can easily be modeled by an MTCCS specification. Furthermore when analyzing systems, one is likely to also constrain the behaviour of the environment of the system to be able to proof desired properties. The graph that results from the transformation has thus only two transition labels: γ and ι . In the rules, additional labels are used that are needed to build communication actions:

- A label $l?(\beta, v, t, d)$, where l is a mtccs synchronization label, β a process identifier, and t, d are variable identifiers
- A label $l!(\beta, e, t, d)$, where e is a value expression

Let $Spec$ be an MTCCS specification. Then $G = (L, l_0, X, T, E)$ is the corresponding expanded graph of which the set of edges E consists of the subset reachable from $Spec.Behaviour$ of the set of edges defined by the smallest set of symbolic transitions satisfying the inference rules given in figure 8. The initial state l_0 corresponds to

³ v is a new variable

⁴ β' is a new PId

⁵The dual rule is not shown

⁶The rule for $l!(e, t, -, c)$ is analogous and therefore not shown

Spec.Behaviour, The set of states L is identified by the set of expressions reachable through the symbolic transition relation defined by E . The set of clocks is defined by the set of process identifiers introduced in the expansion. The form of the store can be derived from the initial behaviour expression.

The SM^* rule defines symbolic transitions associated with state manipulations. An edge is created defining γ actions that perform the state manipulations defined by sm . State manipulations are performed immediately, hence the empty constraint.

$ID1^*$, $ID2^*$, and $ID3^*$ are associated with the idle expression. The first creates a symbolic transition that performs the evaluation of the value expression of the idle expression. It defines an immediately performed ι action that binds a value that results from the evaluation to a newly introduced identifier v . v identifies the variable that holds the amount of time to be delayed. The resulting location is identified by a $Wait^\beta(v, B)$ construct which holds the identifier v , and the behaviour expression B representing the continuation behaviour. The other two rules define the symbolic transitions outgoing from a $Wait^\beta(v, B)$ location. These rules extend outgoing edges of B , by adding a constraint that effects the delay ($CCon(\beta, v)$), and by resetting the local clock ($Res(\beta)$). $ID2^*$ deals with symbolic transitions labeled with γ and ι , $abstID3$ deals with transitions labeled with $l?(\beta, v, t, d)$ constructs. There is analogous rule for the $l!(\beta, e, t, d)$ construct which is not depicted.

Figure 9 is intended to clarify the $ID1^*$ and $ID2^*$ rules. If there is some edge from a location B^1 to a location B'^1

Figure 9: Expansion of Idle constructs

(the upper graph in figure 9), then the lower graph depicts the structure corresponding to $Idle^1(6, B)$. The first edge deals with the evaluation of the value expression ($Loc(1, v, '6')$ denotes the addition of the binding of v to the value of '6' to the local definitions environment identified by 1). The second edge corresponds to the edge from B^1 to B'^1 enhanced with a clock constraint $CCon(1, v)$ (denoting $Clock(1) \geq v$), and a clock reset $Res(1)$. Square brackets are used to represent constraints. Note that 1 is the value of the process identifier which also used to identify the relevant clock.

IN^* and OUT^* are the rules for the input and output actions. An *InAct* location has symbolic transition without update and constraint and an action label $l?(\beta, v, t, 0)$, in which l denotes the action label, β the process identifier, v the input variable, and t the time variable. This kind of edge will never be present in the eventual XTGraph, since we only deal with closed system. For an *OutAct* location analogous symbolic transitions are defined by the OUT^* rule.

The rules for the let construction and the *If* construction, LET^* and $IF1^*$ and $IF2^*$ are straightforward. The latter create two outgoing symbolic transitions from an *If* location, one with a constraint equal to that of the *If* expression, and a second one with the negation of this constraint.

The rules associated with the global and local agent instantiation ($GAGENT^*$, $LAGENT^*$ introduce an update of the current local definitions environment corresponding to the value part.

The rule for the guard construct (GRD^*) simply adds a constraint to symbolic transitions of the construct that is guarded.

$CH1^*$, $CH2^*$, and $CH3^*$ deal with symbolic transitions from *Choice* locations. The first rule effects the splitting of the local definition environment, for which a new process identifier β' is introduced. $CH2^*$ defines symbolic transitions that do not resolve the choice, whereas $CH3^*$ defines those that do.

The next four rules concern the parallel composition. $PAR1G^*$ and $PAR1L^*$ initialize the parallel compositions, and $PAR2^*$ define the transitions associated with non-communication actions $PAR3^*$ deserves some more discussion. The latter is concerned with transitions associated with communications. If the two subexpressions of a parallel composition location have symbolic transitions with matching input and output labels, then this location has a corresponding communication transition. The update belonging to this transition involves the resetting of the clocks and the

addition of at most three local definitions that result from the communication. These are the binding of the communicated value ($Loc(\beta_1, v, e)$), and the binding of the time variables at both sides of the communication. The constraint of the communication transition is the union of the constraints of the input transition and the output transition. The final rules deal with restriction and relabeling and are straightforward.

Figure 10: Three examples of expansion

Figure 10 presents some examples that illustrate the expansion rules. Action labels have been left out, since they are not interesting for closed systems. In the first example, one can see that an MTCCS *let* construct, as well as a *state manipulation* construct results in a single edge labeled with the appropriate environment update. $Loc(1, x, '2')$ denotes adding of the binding of x to the value of $'2'$ to the local definitions environment identified by process identifier 1. $SM(1, 'y := x + 1')$ denotes that the store is updated according to $'y := x + 1'$, which is evaluated in the local definitions environment 1. In the second example the first edge corresponds to the splitting of the current local definitions environment as a result of the parallel composition. The *if* construct results in a location with two outgoing edges, labeled with respectively the condition of the *if* and its negation (square brackets denote constraints). The action with update $Loc(1, y, 'x')$ corresponds to the communication, while the action with update $SM(1, 'y := 1')$ corresponds to the state manipulation. The third example illustrates the transformation of an *idle* expression. The $Loc(3, e, '1')$ and $Loc(2, d, 'y')$ updates are consequences of the evaluations of the value expression of the two *idle* constructs. d and e are new variables introduced by the transformation. There are two *Split* edges, one for the choice construct, and one for the parallel composition. C_i denotes the value of the clock corresponding to process i . The edge corresponding to the communication action (the rightmost bottom edge) has four updates. $Clk(1, t)$ binds the value clock 1 to the time variable t , $Loc(1, x, '2')$ binds the communication value, and the other two reset the clocks of the participating processes.

Finally we will give the expansion of the clock example given in section 2. To obtain a closed specification, an environment is constructed which performs external actions at random times:

Figure 11: An expansion of the clock specification

```

agent Env
ports out reset
in time :  $\mathbb{N}$ 
local
  Env1  $\triangleq$  idle (let  $x \in \mathbb{R}^{\geq 0}$  in  $x$ )  $\odot$   $\overline{Reset}$   $\odot$  Env1
  Env2  $\triangleq$  idle (let  $x \in \mathbb{R}^{\geq 0}$  in  $x$ )  $\odot$  Time( $dum$ )  $\odot$  Env2
Env  $\triangleq$  Env1 | Env2

```

The parallel composition with the example system of figure 3 with the above environment system results in the expansion given in figure 11. For clarity several updates have been omitted from edges. In this case, such an edge has the same labels as other edges in parallel with the edge. Also we have used a more convenient and less precise notation for updates and constraints. u, v , and w are new variables introduced by the transformation, $c1, c2$, and $c3$ are clocks. Again one can see that a lot of superfluous actions are present in the graph. Optimization would lead to a much smaller graph. For example the splits do not need to be modeled by separate actions.

As can be seen in the above examples, the transformation introduces a number of actions that could easily be taken together with other actions. These are often actions that are associated with the evaluation of value expressions. The reason that these evaluations are modeled as separate actions is that the value expressions may contain data elements that could be changed by another parallel process. In practice however, this is often not the case. For example in

let $x = 3$ in $\sigma(\dots) \dots$

the evaluation of the expression can safely be taken together with the subsequent action. There are some more possibilities for optimization. An implementation of the expansion would probably include the following optimizations:

- ι actions associated with idle, if, let and agentinst constructs do not have to be modeled by separate transitions if the value expression they evaluate do not refer to store variables.
- Environment splits do not have to be modeled by separate actions.
- Splitting the environment as a consequence of choice constructs is often not needed. The local environment associated with the branch that is not taken can be discarded.

In [19] a more elaborate discussion of optimizations can be found.

Applying the expansion to a closed MTCCS specification without recursion through the parallel, restriction and relabeling operators, leads to an equivalent XTGraph. The proof can be found in [19].

5 Discussion

We described MTCCS, a formal language for defining real-time concurrent systems with data. MTCCS is based on earlier work on a language called MOSCA [27, 28, 29].

Like for example LOTOS [30] and PSF [20], MTCCS is founded on a combination of a process algebra with a data description formalism. In these process algebras, data is treated in different manner. There an environment is created that keeps track of substitutions introduced by input actions and local definitions. Our approach explicitly models the state of a system. One advantage of this approach is that it has become possible to use shared data structures. The fact that MTCCS allows shared data has some consequences for its semantics. If shared data were not allowed, a number of semantics rules could be simplified, significantly reducing the number of ι actions in the resulting transition system. The ι actions associated with the *idle*, *let*, and *if* constructs would be superfluous. It may seem that this introduces a lot of redundancy in the model, but most of these actions can simply be optimized away, either directly, or indirectly by means of reduction techniques discussed below. Note that by omitting the state component and the state manipulation prefix, one arrives at a value-passing based model.

The time model of MTCCS is based on that of Timed CCS, which is a simple but effective way to model timed behaviour. The fact that MTCCS also incorporates data, increases the power of our time modeling constructs. It allows for example the expression of intervals in delays, and dealing with elapsed time values as normal data values. To describe data manipulations we use VDM-SL. The work described in this paper is to a great extent independent of this choice. Any data manipulation formalism could be used, as long as it adheres to the same interface.

We showed that for a process algebra like MTCCS it is possible formally derive a graph based model that can be used for verification purposes. This work is part of research efforts aimed at arriving at a specification and verification strategy for MTCCS specifications, based on model checking. The main challenge in applying model checking is in avoiding the state space explosion. An optimal model checking strategy focuses on reducing all three sources of this state space explosion; interleaving, timing, and data. Figure 12 presents an overview of what we think such a strategy could look like. Our work is based on a two-phased approach to verification of MTCCS specifications. First,

Figure 12: Overview of the verification approach

the compositional MTCCS specification is translated to a global graph-based representation, and subsequently model checking is applied. The transformation defined in this paper serves as a basis for development of more sophisticated techniques that intend to avoid the explosion that comes with the expansion of parallelism [19]. The expansion defined here fully expands the control space of the compositional process algebra specification. We are currently working on combining the transformation with reduction techniques, resulting in considerably smaller intermediate models, reducing the complexity of the subsequent model checking step. This partial expansion will use reduction techniques based on partial order model checking approaches [9, 24].

The final step in this overview is where symbolic model checking is applied. An example of such a technique is symbolic real-time model checking [12, 26], which focuses on timing aspects and data aspects in the form of linearly changing continuous variables [2]. As stated before our intention is to apply verification to systems with more elaborate data components. Therefore ways have to be found to deal with the state space explosion that is inherent to data elements. One way could be to extend symbolic model checking to other kinds of data components than linearly changing continuous variables, like for example sets. A second approach could be to exploit the symmetry

that is present in many data components. Finally one could also use data abstraction techniques [7], to simplify data components. The disadvantage of such abstraction techniques [7] is that it requires complex human involvement in finding the proper abstractions, which destroys the mechanic character of the model checking approach. However, for certain classes of systems, the use of abstraction techniques may very well be inevitable. For that reason we envision a data abstraction step which simplifies the data component of a system so that symbolic model checking becomes possible.

At present we have developed the mentioned partial expansion technique, and are in the process of implementing it in a tool. As a next step we are interested in attacking the other two sources of complexity, timing and data, by applying symbolic model checking techniques [12, 2, 26] to our reduced XTGraph models.

References

- [1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993.
- [2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
- [3] R. Alur and D. Dill. The theory of timed automata. In *Proceedings REX workshop on Real-Time: Theory and Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 45–73. Springer-Verlag, 1991.
- [4] H. R. Andersen. Partial model checking. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science*, pages 398–407. IEEE Computer Society Press, 1995.
- [5] D. Bosscher and A. Ponse. Translating a process algebra with symbolic data values to linear format. In *Proceedings of the 1st Workshop on Tools and Algorithms for the construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 119–130. Springer-Verlag, 1995.
- [6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
- [7] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
- [8] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proceedings of the 7th International Conference on Formal Description Techniques*, pages 227–242. Chapman and Hall, 1995.
- [9] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial approach to branching time logic model checking. In *Proceedings of the Third Israel Symposium on Theory of Computing and Systems*, pages 130–139. IEEE Computer Society Press, 1994.
- [10] J. F. Groote. Transition system specifications with negative premises. In *Proceedings of Concur'90*, volume 458 of *Lecture Notes in Computer Science*, pages 332–341. Springer-Verlag, 1990.
- [11] M. Hennessy and H. Lin. Symbolic bisimulations. Technical Report 92-1, School of Cognitive and Computing Sciences, University of Sussex, 1992.
- [12] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.
- [13] ISO/IEC JTC1/SC22/WG 19. *VDM-SL, First Committee Draft Standard: CD 13817-1*, 1993.
- [14] C. B. Jones. *Systematic software development using VDM, second edition*. Prentice Hall International, 1990.

- [15] J. van Katwijk, R. F. Lutje Spelberg, S. Stuurman, and W. J. Toetenel. Modeling and analysis of complex computer systems; the MTCCS approach. In *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems*, pages 423–430. IEEE Computer Society Press, 1996.
- [16] A. Kerbrat. Reachable state space analysis of LOTOS specifications. In *Proceedings of the 7th International Conference on Formal Description Techniques*, pages 181–195. Chapman and Hall, 1995.
- [17] K. G. Larsen, P. Petterson, and Wang Yi. Model checking for real-time systems. In *Proceedings of Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer-Verlag, 1995.
- [18] R. F. Lutje Spelberg, S. Stuurman, and W. J. Toetenel. MTCCS. Technical report, Delft University of Technology, Faculty of Technical Mathematics and Informatics, januari 1997. In preparation.
- [19] R. F. Lutje Spelberg and W. J. Toetenel. Partial expansion of real-time process algebra specifications with data into extended timed automata. Technical report, Delft University of Technology, Faculty of Technical Mathematics and Informatics, januari 1997. In preparation.
- [20] S. Mauw and G. J. Veltink. *Algebraic Specification of Communication Protocols*. Cambridge Tracts in Computer Science 36. Cambridge University Press, 1993.
- [21] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.
- [22] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18(9):794–804, 1992.
- [23] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30:181–202, 1993.
- [24] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings 6th International Conference on Computer Aided Verification, CAV'94*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer-Verlag, 1994.
- [25] M. Z. Schreiber. *Value-passing process calculi as a formal method*. PhD thesis, University of London, 1994.
- [26] O. V. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings 7th International Conference on Computer Aided Verification, CAV'95*, volume 939 of *Lecture Notes in Computer Science*, pages 211–224. Springer-Verlag, 1995.
- [27] W. J. Toetenel. *Model Oriented Specification of Communicating Agents*. PhD thesis, Delft University of Technology, 1992.
- [28] W. J. Toetenel. VDM + CCS + TIME = MOSCA. In *Proceedings of the IFIP/IFAC Workshop on Real-time Programming*, 1992.
- [29] W. J. Toetenel. Loose real-time communicating agents. In *Proceedings of the 1st Workshop on Semantics of Specification Languages*, pages 135–151. Springer-Verlag, 1993.
- [30] P. H. J. van Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS*. Elsevier Science Publishers (North-Holland), 1989.
- [31] Wang Yi. CCS + time = an interleaving model for real-time systems. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1991.

A EvalU and EvalC

$EvalU : U \rightarrow (Env \rightarrow Env)\text{-set}$

$EvalU(us) \triangleq$

if $us = \emptyset$

then $\{\lambda env \cdot env\}$

else let $u \in us$ in

let $tr = EvalU(us \setminus u)$ in

if $\text{is-SM}(u)$

then $\{\lambda \text{mk-Env}(\rho, \nu, \psi) \cdot$

let $\text{mk-Env}(\rho', \nu', \psi') = t(\rho, \nu, \psi)$ in

$\text{mk-Env}(\rho', \mu(\rho'(u.pid), \nu'), \psi) \mid \mu \in eval[u.sm], t \in tr\}$

elseif $\text{is-Loc}(u)$

then $\{\lambda \text{mk-Env}(\rho, \nu, \psi) \cdot$

let $\text{mk-Env}(\rho', \nu', \psi') = t(\rho, \nu, \psi)$ in

let $\rho'' = \rho' \dagger \{u.pid \mapsto (\rho(u.pid) \dagger \{u.var \mapsto ev(\rho'(u.pid), \nu')\})\}$ in

$\text{mk-Env}(\rho'', \nu', \psi') \mid ev \in eval[u.exp], t \in tr\}$

elseif $\text{is-Loc}^*(u)$

then $\{\lambda \text{mk-Env}(\rho, \nu, \psi) \cdot$

let $\text{mk-Env}(\rho', \nu', \psi') = t(\rho, \nu, \psi)$ in

let $\rho'' = \rho' \dagger \{u.pid \mapsto \{u.var \mapsto ev(\rho'(u.pid), \nu')\}\}$ in

$\text{mk-Env}(\rho'', \nu', \psi') \mid ev \in eval[u.exp], t \in tr\}$

elseif $\text{is-Clk}(u)$

then $\{\lambda \text{mk-Env}(\rho, \nu, \psi) \cdot$

let $\text{mk-Env}(\rho', \nu', \psi') = t(\rho, \nu, \psi)$ in

let $\rho'' = \rho' \dagger \{u.pid \mapsto (\rho'(u.pid) \dagger \{u.var \mapsto \psi'(u.pid)\})\}$ in

$\text{mk-Env}(\rho'', \nu', \psi') \mid t \in tr\}$

elseif $\text{is-Res}(u)$

then $\{\lambda \text{mk-Env}(\rho, \nu, \psi) \cdot$

let $\text{mk-Env}(\rho', \nu', \psi') = t(\rho, \nu, \psi)$ in

$\text{mk-Env}(\rho', \nu', \psi' \dagger \{u.pid \mapsto 0\}) \mid t \in tr\}$

elseif $\text{is-Split}(u)$

then $\{\lambda \text{mk-Env}(\rho, \nu, \psi) \cdot$

let $\text{mk-Env}(\rho', \nu', \psi') = t(\rho, \nu, \psi)$ in

let $\rho'' = \rho' \dagger \{u.pid' \mapsto \rho'(u.pid)\}$ in

$\text{mk-Env}(\rho'', \nu', \psi') \mid t \in tr\}$

$EvalC : C \rightarrow (Env \rightarrow \mathbb{B})$

$EvalC(cs) \triangleq$

$\lambda \text{mk-Env}(\rho, \nu, \psi) \cdot$

if $cs = \emptyset$

then true

else let $c \in cs$ in

$EvalC(cs \setminus \{c\})(\rho, \nu, \psi) \wedge$

if $\text{is-CCon}(c)$

then $\psi(c.pid) \geq \rho(h.pid)(h.var)$

else let $ev \in eval[c.con]$ in $ev(\rho(c.pid), \nu)$