Modeling and analysis of complex computer systems – the MTCCS approach –

Hans Toetenel, Ronald Lutje Spelberg, Sylvia Stuurman, Jan van Katwijk
Faculty of Technical Mathematics and Informatics
Delft University of Technology
132 Julianalaan, 2628 BL Delft, The Netherlands

Abstract

The paper presents results from work in progress on finding a method for formal specification and verification of real-time concurrent systems that incorporate a non trivial data component. We have extended Timed CCS, a timed CCS variant with a model-oriented data language based on VDM. The semantics of the extension, called MTCCS is expressed in a combination of denotational and operational style. We show how verification of temporal logic properties based on symbolic model checking can be made possible for such a combination notation.

1 Introduction and overview

Complex computer systems are frequently highly concurrent, distributed and often have real-time properties. Since Leveson [14] had pointed out that *modeling and analysis* form the main challenges in building complex real-time systems much research has been carried out in the field of formal specification. The underlying theory of formal specification techniques has been investigated thoroughly and can be regarded as relatively mature. At the same time, tools have been developed that facilitate construction and analysis of formal specifications. The benefits of the use of formal specification languages are clear. They provide a concise framework within which software requirements and designs can be expressed unambiguously. The resulting specifications are more suited for analysis and verification.

Figure 1 schematically depicts relationships between notations and tools we use in a notational framework for formal system modelling, analysis and implementation.

MTCCS, (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 [23], a timed version of process algebra notation CCS [16] and VDM-SL [1] [8], the specification notation of the VDM methodology [5], [13].

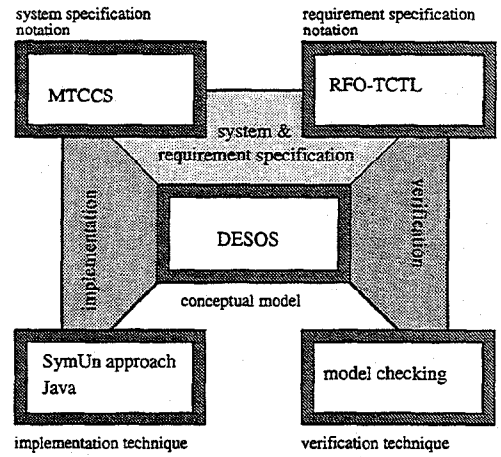

Figure 1: Notational Framework

To express properties of MTCCS specifications we us a real-time temporal logic based on TCTL [2].

The conceptual model is based on a structural integration of denotational semantics and structural operational semantics (DESOS). In this approach the overall semantic style is operational. In the definition of the transition rules semantic functions are applied which are defined by denotational semantic definitions. Both MTCCS and RFO-TCTL are given a DESOS semantics.

The implementation of MTCCS specifications is based on SYM-UN, a distributed tree based communication protocol for basic CCS implemented in the **Java** language. Its definition and implementation of

[1] The specification language for VDM for which an ISO standard is currently being developed (ISO SC22/WG19/N-20).

Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)
0-8186-7614-0/96 $10.00 © 1996 IEEE

the SYM-UN protocol are omitted from this paper.

Our method for verifying property specifications is based on model checking, which allows verification of temporal logic properties in a highly automatic manner.

The remainder of this paper is organized as follows. In section 2 an overview is presented of the MTCCS. Section 3 introduces the notation for requirement specification RFO-TCTL. Section 4 is dedicated to verification. It summarizes our approach to model-checking MTCCS specifications with respect to properties expressed in the RFO-TCTL notation. Fimally section 5 presents a discussion of our results sofar, compares our approach to related work and projects our work into the future. 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 introduces shortly the syntax and semantics of MTCCS. For a more comprehensive description of the language, see [15]. MTCCS is a succesor to a syntactic and semantic framework based on the same combination of VDM-SL and TCCS, called MOSCA [22]. The aim of the development of MOSCA was to increase the applicability of VDM in the area of distributed, parallel and real-time systems. It has been given various forms of semantics. The current form of the MTCCS semantics is strongly based on the first semantics for MOSCA given in [21]. The main differences between MOSCA and MTCCS are the more precise definition of the state component of MTCCS and the particular approach to model the looseness aspect of VDM-SL within the operational semantics of MTCCS.

A 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 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.

The core syntax of the process component of MTCCS is defined by the abstract domains given in figure 2. It forms the base syntax on which the MTCCS semantics are defined. The core syntax of VDL-SL is omitted in this paper. The full definition of the core syntax of VDM-SL can be found in [1].

$$
\begin{array}{l}
Spec :: Behaviour : ProcId \xrightarrow{m} (BExpr \times VarId) \\
\qquad State : ProcId \xrightarrow{m} (VarId \xrightarrow{m} Type)) \\
\qquad Init : BExpr \\
BExpr = InAct \mid OutAct \mid Idle \mid StateMan \mid Let \mid If \mid \\
\qquad Agent \mid Choice \mid Par \mid Restrict \mid Relabel \mid Null \\
\qquad Wait \mid Delay \\
InAct :: label : ActId \\
\qquad mvar : VarId \\
\qquad dvar : VarId \\
\qquad cont : BExpr \\
OutAct :: label : ActId \\
\qquad mval : VExpr \\
\qquad dvar : VarId \\
\qquad cont : BExpr \\
Idle :: durval : VExpr \\
\qquad cont : BExpr \\
StateMan :: man : Statement \\
\qquad cont : BExpr \\
Let :: var : VarId \\
\qquad val : VExpr \\
\qquad cont : BExpr \\
If :: cond : VExpr \\
\qquad cont : BExpr \\
\qquad alt : BExpr \\
Agent :: name : ProcId \\
\qquad val : [VExpr] \\
Choice :: left : BExpr \\
\qquad right : BExpr \\
Par :: left : BExpr \\
\qquad right : BExpr \\
Restrict :: res : Id\text{-set} \\
\qquad cont : BExpr \\
Relabel :: cont : BExpr \\
\qquad rel : ActId \xrightarrow{m} ActId \\
Wait :: timer : \mathbb{R}^{\geq 0} \\
\qquad cont : BExpr \\
Delay :: act : InAct \mid OutAct \\
\qquad delay : \mathbb{R}^{\geq 0}
\end{array}
$$

Figure 2: MTCCS Core syntax

The core syntax domain *VExpr* denotes all value expressions defined by the VDM-SL part of MTCCS. It offers basic types like natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), reals ($\mathbb{R}$), booleans ($\mathbb{B}$), 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. It features subtyping through type invariants. Some concrete syntax constructions to specify the behaviour of agents are shown in Figure 2. The core domain *Spec* holds the information of defined agent behaviours (*Behaviour*), global state components (*State*), and the behaviour associated with the specification (*Init*). The first four behaviour expression domains *InAct*, *OutAct*, *Idle*, and *StateMan* form the prefix expressions. The first two prefix behaviour expression domains, *InAct* and *OutAct* are the timed prefixes of TCCS. The third prefix domain describes the idle prefix of TCCS, which expresses time progression. The fourth prefix domain handles the infusion of

Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)
0-8186-7614-0/96 $10.00 © 1996 IEEE

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

Figure 3: MTCCS concrete syntax examples

state manipulation. It specifies a VDM-SL statement that manipulates the state of the agent in which the construction appears.

The *If* expression enables conditional behaviour specification steered by data conditions. The *Let* enables local value bindings, either fixed or loose. E.g. in

$$\text{let } x = 3 \text{ in } A\langle x \rangle$$

the value 3 is bound to the identifier x in a deterministic way. In

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

it is not deterministically decided what the value of the expression bound to x will be. It may be either 1, or 2 or 3. Depending on the context of the expression the choice is made in specification time or in execution time. The *Agent* domain describes parameterized agent instantiation, like $A\langle x \rangle$. The next four domains describe standard CCS operators. The *choice* construction enables nondeterministic selection of specific behaviour.

The *Choice*, *Par*, *Restrict*, and *Relabel* domains model the standard CCS constructs. The first specifies non-deterministic choice between behaviours, while the other three are used to model parallelism and communication.

The *Null* construction denotes the passive agent null. *Wait* and *Delay* are two core syntax domains without a concrete syntax counter part. They are intermediate forms, used to give meaning to the passing of time.

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 exact form of the data manipulation part of MTCCS is not discussed in this paper, but can be found in [1]. 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 $\langle B, \rho, \nu \rangle \in (BExpr \times EnvT \times Store)$ consists of three components, an MTCCS behaviour expression B, a local definitions valuation ρ, and a state valuation ν.

EnvT and *Store* are defined as follows.

$$Env, Store = Id \xrightarrow{m} Val$$

$$EnvT = EnvT \cdot EnvT \mid Env$$

A local definitions environment $\rho \in EnvT$ is tree of environments, patterned after the process structure of the corresponding global state. 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 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. There is one global state environment $\nu \in Store$, which is shared by all subprocesses.

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:

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

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:

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

[2]The dual rule is not shown

Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)
0-8186-7614-0/96 $10.00 © 1996 IEEE

Spec	$Spec(bh, st, init) \xrightarrow{\iota} \langle init, \{vid \mapsto \text{-} \mid vid \in \mathsf{dom} \bigcup \mathsf{rng}\ st\}, \emptyset \rangle$
StateMan	$[\mu \in eval[\![sm]\!]]\ \langle StateMan(sm, B), \rho, \nu \rangle \xrightarrow{\gamma} \langle B, \rho, \mu(\rho, \nu) \rangle$
Idle1	$[d = ev(\rho, \nu),\ ev \in eval[\![e]\!]]\ \langle Idle(e, B), \rho, \nu \rangle \xrightarrow{\iota} \langle Wait(d, B), \rho, \nu \rangle$
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	$\langle act, \rho, \nu \rangle \xrightarrow{\iota} \langle Delay(act, 0), \rho, \nu \rangle\ \ act \in \{InAct(\text{-},\text{-},\text{-},\text{-}), OutAct(\text{-},\text{-},\text{-},\text{-})\}$
ActDelay2	$\langle Delay(act, d), \rho, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle Delay(act, d + \delta), \rho, \nu \rangle\ \ act \in \{InAct(\text{-},\text{-},\text{-},\text{-}), OutAct(\text{-},\text{-},\text{-},\text{-})\}$
InAct	$\langle Delay(InAct(l, v, t, B), d), \rho, \nu \rangle \xrightarrow{l?val} \langle B, \rho[v \mapsto val, t \mapsto d], \nu \rangle$
OutAct	$[ev \in eval[\![e]\!],\ val = ev(\rho, \nu)]\ \langle Delay(OutAct(l, e, t, B), d), \rho, \nu \rangle \xrightarrow{l!val} \langle B, \rho[t \mapsto d], \nu \rangle$
Let	$[\rho' = \rho[v \mapsto ev(\rho, \nu)],\ ev \in eval[\![e]\!]]\ \langle Let(v, e, B), \rho, \nu \rangle \xrightarrow{\iota} \langle B, \rho', \nu \rangle$
Agent	$[Agent(P) = (B, v), \rho' = \{v \mapsto ev(\rho, \nu)\}, ev \in eval[\![e]\!]]\ \langle AgentInst(e, P), \rho, \nu \rangle \xrightarrow{\iota} \langle B, \rho', \nu \rangle$
If1	$[ev \in eval[\![cond]\!],\ ev(\rho, \nu) = \text{TRUE}]\ \langle AgentIf(cond, B_1, B_2), \rho, \nu \rangle \xrightarrow{\iota} \langle B_1, \rho, \nu \rangle$
If2	$[ev \in eval[\![cond]\!],\ ev(\rho, \nu) = \text{FALSE}]\ \langle AgentIf(cond, B_1, B_2), \rho, \nu \rangle \xrightarrow{\iota} \langle B_2, \rho, \nu \rangle$
Choice1	$\langle Choice(B_1, B_2), \rho, \nu \rangle \xrightarrow{\iota} \langle Choice(B_1, B_2), \rho \cdot \rho, \nu \rangle$
Choice2	$\dfrac{\langle B_i, \rho, \nu \rangle \xrightarrow{x} \langle B_i', \rho', \nu' \rangle,\ x \notin \{\epsilon(\text{-}), \iota\}}{\langle Choice(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{x} \langle B_i', \rho_i', \nu \rangle}\ i \in \{1, 2\}$
Choice3	$\dfrac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B_1', \rho_1, \nu \rangle,\ \langle B_2, \rho_2, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B_2', \rho_2, \nu \rangle}{\langle Choice(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle Choice(B_1', B_2), \rho_1' \cdot \rho_2', \nu \rangle}$
Choice4	$\dfrac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{\iota} \langle B_1', \rho_1', \nu \rangle}{\langle Choice(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\iota} \langle Choice(B_1', B_2), \rho_1' \cdot \rho_2, \nu \rangle}\ 2$
Par1	$\langle Par(B_1, B_2), \rho, \nu \rangle \xrightarrow{\iota} \langle Par(B_1, B_2), \rho \cdot \rho, \nu \rangle$
Par2	$\dfrac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{l!val} \langle B_1', \rho_1', \nu \rangle,\ \langle B_2, \rho_2, \nu \rangle \xrightarrow{l?val} \langle B_2', \rho_2', \nu \rangle}{\langle Par(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\gamma} \langle Par(B_1', B_2'), \rho_1' \cdot \rho_2', \nu \rangle}$
Par3	$\dfrac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B_1', \rho_1, \nu \rangle,\ \langle B_2, \rho_2, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle B_2', \rho_2, \nu \rangle,\ \langle Par(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \not\xrightarrow{\gamma}}{\langle Par(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{\epsilon(\delta)} \langle Par(B_1', B_2'), \rho_1 \cdot \rho_2, \nu \rangle}$
Par4	$\dfrac{\langle B_1, \rho_1, \nu \rangle \xrightarrow{x} \langle B_1', \rho_1', \nu' \rangle,\ x \neq \epsilon(\text{-})}{\langle Par(B_1, B_2), \rho_1 \cdot \rho_2, \nu \rangle \xrightarrow{x} \langle Par(B_1', B_2), \rho_1' \cdot \rho_2, \nu' \rangle}\ 2$
Restrict	$\dfrac{\langle B, \rho, \nu \rangle \xrightarrow{x} \langle B', \rho', \nu' \rangle,\ x \notin res}{\langle Restrict(res, B), \rho, \mu \rangle \xrightarrow{x} \langle Restrict(res, B'), \rho', \nu' \rangle}$
Relab	$\dfrac{\langle B, \rho, \nu \rangle \xrightarrow{x} \langle B', \rho', \nu' \rangle}{\langle Relab(rel, B), \rho, \nu \rangle \xrightarrow{x[rel]} \langle B', \rho', \nu' \rangle}$
Null	$\langle Null, \rho, \nu \rangle \xrightarrow{\epsilon(d)} \langle Null, \rho, \nu \rangle$

Figure 4: Semantics of MTCCS

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 state environment and returns a new state environment. In the case a value expression (*VExpr*) is evaluated a set of evaluators is returned. Each evaluator takes a local and a state environment and returns a value.

An environment that holds the 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 *Agent* rule. In the latter the existence of a function *Agent* : $ProcId \xrightarrow{m} (BExpr \times VarId)$ is assumed.

For clarity we will 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.

3 Requirement specification

The requirement specification notation is intimately associated with the approach taken for verification, in our case model checking. The temporal logics traditi-

Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)
0-8186-7614-0/96 $10.00 © 1996 IEEE

onally used in model checking are propositional logics, since these technique abstract from the data domain by associating with each state a set of atomic propositions true in that state. Since our model does incorporate a state, we will use a first order logic, in our case an extension of TCTL (Timed Computation Tree Logic) [2] called RFO-TCTL (Restricted First-Order TCTL). The logic is called restricted because quantification over temporal operators is not allowed, which makes our extension quite straightforward. Atomic propositions are simply replaced by boolean expressions from our data manipulation language. The following grammar defines the syntax of RFO-TCTL:

$$\begin{aligned}\phi &::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \\ &\quad A(\phi_1 U \phi_2) \mid E(\phi_1 U \phi_2) \mid z.\phi \mid \psi \\ \psi &::= z_1 + c_1 \leq z_2 + c_2 \mid \neg\psi \mid \psi_1 \vee \psi_2\end{aligned}$$

where z, z_1, z_2 range over a set of specification clocks, and $c_1, c_2 \in \mathbb{R}^{\geq 0}$. p ranges over the set of (boolean) expressions of our data manipulation language. p is satisfied if the valuation of the store satisfies the boolean value expression p. $A(\phi_1 U \phi_2)$ is satisfied if for all computation paths starting from the current state, there is a state along it which satisfies ϕ_2, and until that time ϕ_1 is satisfied. $E(\phi_1 U \phi_2)$ is satisfied if there is at least one such computation path. $z.\phi$ is satisfied by the current state if ϕ is satisfied by the state obtained from the current state by introducing a new specification clock z which is set to zero. ψ is satisfied if the values of the specification clocks satisfy ψ. Some derived operators are: $EF\phi$ (There is path on which there is state satisfying ϕ), $EG\phi$ (There is a path of which every state satisfies ϕ), $AF\phi$ (On all paths there is some state satisfying ϕ), $AG\phi$ (On all paths every state satisfies ϕ). A more detailed description and a formal semantics can be found in [20].

As an example $z.EF(p \wedge z < 1)$ expresses that there is within one time unit a future state which satisfies p. $AG(z.(p \rightarrow (AFq \wedge z < 2))$ expresses that whenever some state satisfies p, then q is always satisfied within 2 time units.

4 Verification

To verify MTCCS specifications we apply *model checking*, an algorithmic approach to verification that allows fully automatic verification of temporal logic propertie specifications. In the verification of real-time and hybrid systems, symbolic (real-time) model checking techniques [12, 19] have been quite succesful. These model checking approaches work on representations that in two ways different from our MTCCS formalism. Firstly these representations are graph-based formalisms like for example timed automata. Secondly, symbolic model checking techniques operate on *global* models instead of compositional models.

Our verification method is therefore based on a two-phased approach to verification of MTCCS specificatiions. First, the compositional MTCCS specification is translated to a global graph-based representation, and subsequently model checking is applied. This approach has already been followed for translating several real-time process algebras without data to timed automata variants [17, 9]. However, these are relatively straightforward techniques as they translate a process algebra specification by fully expanding its control space. We combine the transformation from a process algebra specification to a graph representation with a reduction technique to obtain smaller intermediate models, reducing the complexity of the subsequent symbolic model checking step. The intermediate graph representation, called XTGraphs (extended timed graphs) is a variant of timed automata enhanced to symbolically represent data components. The expansion preserves the properties of our requirement specification language RFO-TCTL, so that the resulting model can be used for model checking purposes.

Timed automata [4] are finite state transition graphs equipped with a finite set of continuous clocks all running 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 further extended with capabilities to operate on data, by adding data manipulations and data constraints to transitions, and adding sorts to locations.

An XTGraph is a tuple (S, s_0, T, X, E), where

- S is a set of Locations, with initial location s_0
- $T : S \rightarrow Sort$ assigns a sort to each location
- X is a set of clocks
- E is a set of edges: $E \subseteq S \times S \times L \times U \times C$

where L is a set of action labels, C the set of constraints on clocks and data, and U set of state updates (clocks resets and data manipulations). The semantics of time is somewhat different from that of the timed automata variants usually used in model checking. We focus on verification of *closed* MTCCS specifications. In such a specification an enabled action is always performed immediately (that is, without letting time pass). This is reflected in the semantics of our XTGraphs representation: Time may only pass in some location if none of the outgoing transitions are enabled. Other timed automata variants do not force the execu-

Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)
0-8186-7614-0/96 $10.00 © 1996 IEEE

tion of transitions in this way. Timed safety automata [12] for example equip locations with clock constraints, to force the execution of transitions. Restricting ourselves to defining closed systems does not reduce the expressiveness of MTCCS, since the non-deterministic environment that comes with an open system can easily be modeled by MTCCS processes.

The XTGraph given in figure 5 corresponds to the example system given in specification ?? in section 2. Note that this is not a complete XTGraph, since it does not model a closed system (the *time* and *reset* actions are external). We constructed a translation from MTCCS to XTGraphs that results in an XTGraph that exhibits exactly the same behaviour as the original MTCCS model. This translation involves full expansion of the paralellism present in the specification, introducing an exponential explosion in the size of the state space. The expansion of parallel processes introduces many different interleavings whose distinction is irrelevant since they satisfy the same RFO-TCTL specifications. That is why we extended our transformation with a reduction technique, which is based on partial order model checking techniques [11, 18, 10].

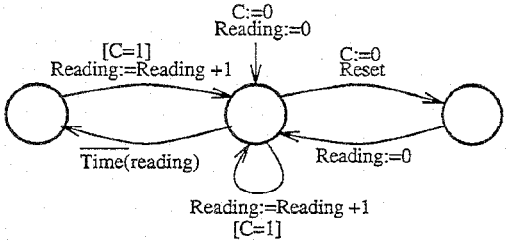

Figure 5: Example XTGraph

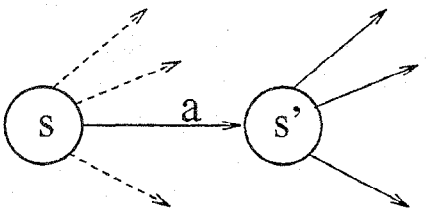

Figure 6: Reduction

For many states of a system it is safe to only consider a subset of all its outgoing transitions and successor states, rather than all of them, avoiding unnecessary enumeration of trivially different interleavings. Reduction is based on the fact that if two states s and it successor s' have the same 'observed' behaviour, then it suffices to consider only the transition a leading from s to s', ignoring all other transitions outgoing from s (see figure 6). The 'ignored' transitions can then safely be postponed to the next state. Whether or not two states have the same observed behaviour is dependent of the property specification to be verified. The partial expansion is based on a depth first search algorithm. During transformation, in each state the possible outgoing transition are examined, but instead of interpreting each transition as an edge in the graph model, transitions are omitted if possible.

Consider a simple example fragment defined by the following behaviour expression:

$$\sigma(x := 4) \odot \mathsf{idle}(2) \odot \overline{a}(x) \odot \cdots \mid (\mathsf{let}\ v = \mathit{SomeValue}\ \mathsf{in}\ (a(y) \star t \odot \cdots \oplus \mathsf{Idle}(v) \odot \sigma(y := 0) \odot \cdots)$$

The corresponding XTGraph is given in figure 7. ($C1$ and $C2$ are clocks, d and e are variables introduced by the transformation). When applying only a partial expansion, an XTGraph is constructed that only contains the solid-lined transitions and locations. Even for this simple example a considerable reduction is achieved.

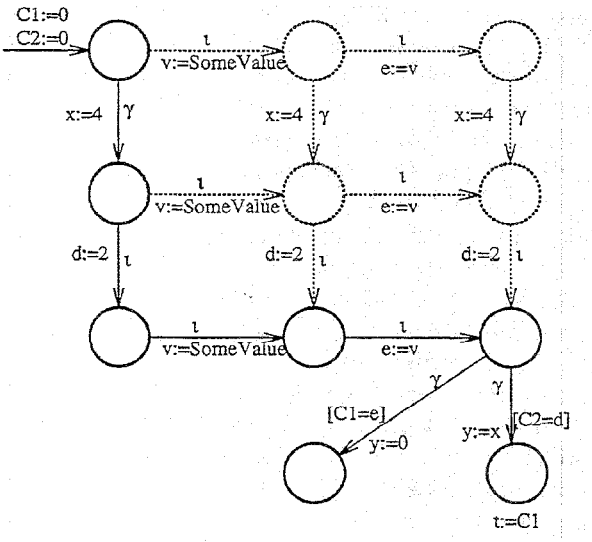

Figure 7: Full and partial expansion

Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)
0-8186-7614-0/96 $10.00 © 1996 IEEE

5 Discussion

We described MTCCS, a formal language for defining real-time concurrent systems with data. Like for example LOTOS [6], 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 the 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 extend 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 of parallel processes, timing, and data. Figure 8 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, 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 [20]. 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 two reduction techniques. The first [20] is based on partial order model checking techniques [10, 18]. We currently working on a second reduction technique based on exploiting symmetry present in system specifications.

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], which focuses on timing aspects and data aspects in the form of linearly changing continuous variables [3]. Our intention is to also apply these symbolic techniques to the other kinds of data components. For complex data aspects the application of symbolic techniques may very well not be feasible. For that reason we envision a data abstraction step which simplifies the data component of a system so that symbolic model checking becomes possible. 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 seems inevitable. Data-intensive systems are very hard to deal with using only the discussed automatic model checking techniques.

The partial expansion technique is focused on reducing the complexity on the control space of the system. As a next step we interested in attacking the other two sources of complexity, timing and data, by applying symbolic model checking techniques [12, 3, 19] to our reduced XTGraph models.

At present we have developed the mentioned partial expansion technique, and are in the process of implementing it in a tool.

References

[1] Vdm-sl, first committee draft standard: Cd 13817-1. Technical report, ISO/IEC/JTC1/SC22/WG 19, November 1993.

[2] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104:2-34, 1993.

Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)
0-8186-7614-0/96 $10.00 © 1996 IEEE

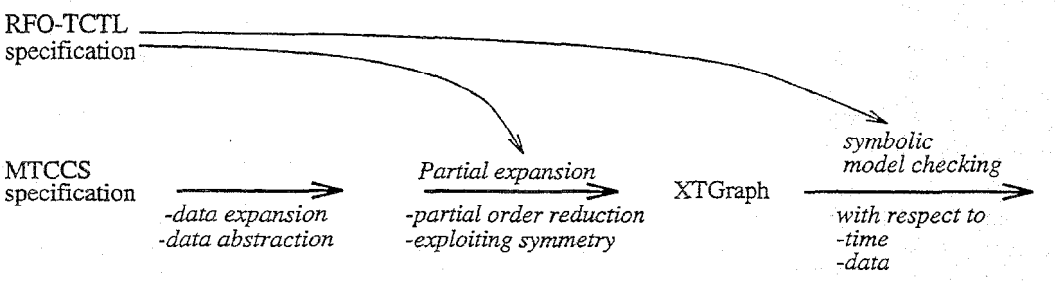

Figure 8: Overview of the verification approach

[3] 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.

[4] 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.

[5] D. Bjørner and C.B. Jones. *Formal Specification & Software Development*. PHI. Prentice Hall, 1982.

[6] T. Bolognesi and E. Brinksma. Introduction to the iso specification language lotos. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal description Technique LOTOS*, pages 23–77. North Holland, 1989.

[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] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[9] 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.

[10] 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.

[11] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110:305–326, 1994.

[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] C.B. Jones. *Systematic Software Development Using VDM, 2-nd edition*. PHI. Prentice Hall, 1990.

[14] N.G. Leveson. The challence of building process-control software. *IEEE Software*, pages 55–62, November 1990.

[15] R. F. Lutje Spelberg, S. Stuurman, and W. J. Toetenel. MTCCS. Technical report, Delft University of Technology, Faculty of Technical Mathematics and Informatics, 1996. In preparation.

[16] R. Milner. *Communication and Concurrency*. PHI. Prentice Hall, 1989.

[17] X. Nicollin, J. Sifakis, and S. Yovine. From atp to timed graphs and hybrid systems. *Acta Informatica*, 30:181–202, 1993.

[18] 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.

[19] 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.

[20] 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, 1996.

[21] W.J. Toetenel. *Model Oriented Specification of Communicating Agents*. PhD thesis, Faculty of Technical Mathematics and Informatics, Delft University of Technology, 1992.

[22] W.J. Toetenel. VDM + CCS + TIME = MOSCA. In *Proceedings of the 18th workshop of IFIP/IFAC WRTP'92*. Brugge, 1992.

[23] Y. Wang. Real-time behaviour of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90 Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, pages 502–520. Springer Verlag, 1990.

Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '96)
0-8186-7614-0/96 $10.00 © 1996 IEEE