

# From $\chi_t$ to $\mu$ CRL: Combining Performance and Functional Analysis

Anton Wijs

CWI, Department of Software Engineering,  
P.O.Box 94079, 1090 GB Amsterdam, The Netherlands  
wijs@cwi.nl

Wan Fokkink

Vrije Universiteit Amsterdam, Department of Computer Science,  
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands  
wanf@cs.vu.nl

## Abstract

*In this paper we first give short overviews of the modelling languages timed  $\chi$  ( $\chi_t$ ) and  $\mu$ CRL. Then we present a general translation scheme to translate  $\chi_t$  specifications to  $\mu$ CRL specifications. As  $\chi_t$  targets performance analysis and  $\mu$ CRL targets functional analysis of systems, this translation scheme provides a way to perform both kinds of analysis on a given  $\chi_t$  system model. Finally, we give an example of a  $\chi_t$  system and show how the translation works on a concrete case study.*

## 1 Introduction

Performance analysis is traditionally based on techniques such as simulation, Markov chains and queueing networks. By contrast, main approaches for verifying functional properties are model checking, where temporal formulas are validated by means of an explicit state space search, and theorem proving, which is largely based on axiomatic reasoning at the symbolic level.

In two earlier papers, the specification language LOTOS [10] was used for performance analysis. LOTOS is a process algebraic language with abstract data types, which was originally designed for functional analysis. Hermanns and Katoen [23] verified performance properties of a LOTOS specification of a telephone system; Garavel and Hermanns [19] introduced a general approach to carry out performance analysis within the framework of LOTOS. They introduce timing information into a LOTOS specification, expressing that certain events are delayable by some random delay, captured by an exponential distribution. From this extended LOTOS specification they generate an interactive Markov chain, which is basically a labelled transition

system containing both actions and positive reals as labels, where the positive reals denote delays. They explain how the CADP toolset [15], which is actually meant for functional verification of LOTOS specifications, can be used to also carry out performance analysis with respect to interactive Markov chains. Although the approach of Garavel and Hermanns is promising, it is difficult if not impossible to apply full-blown performance analysis techniques in a functional verification formalism like LOTOS.

In this paper we propose another approach to bridge the gap between performance and functional analysis. Similar to Garavel and Hermanns, we exploit the fact that specification languages for performance and functional analysis tend to have a lot in common, so that a translation from one specification language to the other is quite feasible. However, we propose to keep the performance and the functional analysis separate, in environments targeted to these analyses. Thus we are in principle able to carry out full-blown performance as well as functional analysis.

Our work is closest in spirit to TwoTowers [6], which is a tool that combines performance and functional analysis. It has a single input language, based on the stochastic process algebra EMPA [5]. Performance analysis is based on simulation and reward Markov chains, while functional analysis is performed by the symbolic model checker nuSMV [14].

$\chi$  [2] is a modelling language for the specification of discrete-event, continuous or combined, so-called *hybrid*, systems. It is based on the process algebra CSP [24], and contains some predefined data types. It targets performance analysis of timed systems by means of simulation techniques to estimate throughput and cycle time. A subset of the language  $\chi$ , restricted to specify only discrete-event systems, is called timed  $\chi$ , or  $\chi_t$ . Currently there are no tools available for using the language  $\chi$  (they are being developed), but predecessors of the language and their simula-

tors have been successfully applied to a large number of industrial cases, such as an integrated circuit manufacturing plant, a brewery and process industry plants, see e.g. [1].

$\mu\text{CRL}$  [17] is a modelling language for the specification of discrete-event systems. It is based on the process algebra ACP [4], extended with abstract data types [27]. It targets functional analysis of distributed systems and communication protocols, by means of simulation, model checking and theorem proving. The verification environment of  $\mu\text{CRL}$  together with the model checker CADP, which can serve as a back-end to  $\mu\text{CRL}$ , have been used to analyse for instance an in-flight data acquisition unit [18] and a distributed system for lifting trucks [22]. Moreover, a homegrown theorem prover has been developed for  $\mu\text{CRL}$  [16].

Recently, in [11] the  $\chi_t$  specification of a turntable [12] was translated to three different specification formalisms: UPPAAL, SPIN and  $\mu\text{CRL}$ . While translating to  $\mu\text{CRL}$ , it was concluded, that  $\chi_t$  and  $\mu\text{CRL}$  are quite closely related, and the development started of a general translation scheme from  $\chi_t$  to  $\mu\text{CRL}$ . A general translation is feasible, because, although the modelling languages  $\chi_t$  and  $\mu\text{CRL}$  have different aims, there are some strong similarities. Most importantly, their input languages are both based on process algebra, and they are both action-based.

In this paper we present a general translation from  $\chi_t$  specifications to *linear process equations* [7], which are basically  $\mu\text{CRL}$  specifications without parallelism and communication. The LPE format is important for the  $\mu\text{CRL}$  toolset, because it is used for the internal representation of processes. The translation is inspired by the translation of the turntable in [11]. Note that we have to limit ourselves to translating  $\chi_t$  instead of the complete hybrid  $\chi$ , because  $\mu\text{CRL}$  cannot cope with continuous events. Furthermore, this paper is an extended abstract, meaning that this paper only presents a selection of the most basic atomic actions and operators in  $\chi_t$  and therefore does not provide the complete translation scheme. The complete scheme can be found in the full version of this paper [31]. The verification of a turntable system in [11] illustrates how our translation scheme can be used to combine performance and functional analysis on a real-life case study.

This paper is set up as follows. The next two sections provide a short introduction to  $\chi_t$  and  $\mu\text{CRL}$ : The basics of the languages are listed and a brief explanation is given. Section 4 provides a way to translate  $\chi_t$  processes to linear process equations. Finally section 5 provides an example of translating a  $\chi_t$  model to a  $\mu\text{CRL}$  model. A larger example can be found in [11].

As future work, we plan to first manually apply the translation to larger examples, to gain further confidence in its applicability and improve it where necessary. Then we could work out a correctness proof of the transformation, to guarantee that it preserves a large class of interesting

properties. We also intend to implement the translation and use it to automatically translate  $\chi_t$  specifications of real-life systems to  $\mu\text{CRL}$ . We could then apply the verification environments of  $\chi_t$  (simulation) and  $\mu\text{CRL}$  (model checking and theorem proving) to such examples, thus obtaining the desired combination of performance and functional analysis.

## 2 The language $\chi_t$

The  $\chi$  language was designed as a hybrid modelling and simulation language. Since we are interested only in discrete-event models and verification, we present here just a part of the language, disregarding features that are used for simulation and to model hybrid behaviour. This (discrete-event) subset of the language is known as *timed  $\chi$*  or  $\chi_t$ . For a complete reference of  $\chi$ , see [2].  $\chi_t$  is described in [3].

**Data types.** The  $\chi_t$  language is statically strongly typed. Every variable has a type which defines the allowed operations on that variable. The basic data types are boolean, natural, integer and real number. The language provides a mechanism to build sets, lists, array tuples, record tuples, dictionaries, functions, and distributions (for stochastic models). Channels also have a type that indicates the type of data that is communicated via the channel.

**Time model.** Time in  $\chi_t$  is dense, i.e. timing is measured on a continuous time scale. The weak time determinism principle, or sometimes called the time factorisation property (time doesn't make a choice), and urgent communication (a process can delay only if it cannot do anything else) are implicit. Time additivity (if a process can delay first  $t_1$  and then immediately following  $t_2$  time units, then it can delay  $t_1 + t_2$  time units from the start) is not present. Delaying is enforced by the delay process, but some atomic processes can also implicitly delay.

**Communication model.** Communication in  $\chi_t$  is synchronous, meaning that a *send* and a *receive* action on the same channel cannot happen individually but only together, as one communication action.

**Atomic processes.** The atomic processes of  $\chi_t$  are process constructors and they cannot be split into smaller processes. Now we will present the atomic processes. Due to space limitations we restrict send and receive actions to one value. In reality they can also work with no values or multiple values at the same time, but in practice it is less common. A translation of these more general send and receive processes can be found in the full version of this article [31].

1. The multi-assignment process ( $x_n := e_n$ ). It assigns the values (must be defined) of expressions  $e_1, \dots, e_n$  to the variables  $x_1, \dots, x_n$ , respectively. It does not have the possibility to delay.
2. The skip process. It performs the internal action  $\tau$  and cannot delay.
3. The send process ( $h ! e$ ). It sends the value of the expression  $e$  via channel  $h$ . The value of  $e$  must be defined and of the right type. It is able to delay arbitrarily long.
4. The receive process ( $h ? x$ ). It receives a value via the channel  $h$  and assigns it to the variable  $x$  which must be of the right type. It is also able to delay arbitrarily long.
5. The delay process ( $\Delta t$ ). It delays a number of time units equal to the value of the expression  $t$ . The value of  $t$  must be a positive real number.

**Operators.** Atomic processes can be combined by means of operators. We present a selection of them together with their (informal) semantics. The rest is omitted from this extended abstract. Some operators are omitted due to space limitations and the fact that those operators are less commonly used. Those can be found in the full version of this article [31]. Besides that we do not consider operators that are only used for the definition of the semantics of  $\chi_t$ , since those never appear in specifications. Two exceptions to this are the encapsulation operator and the urgent communication operator. These operators are implicitly used in  $\chi_t$ , but should be considered explicitly when translating a specification to  $\mu\text{CRL}$ .

1. The guard operator ( $\rightarrow$ ). For action behaviour, a process  $b \rightarrow p$  behaves as  $p$  if the value of the boolean expression (guard)  $b$  is *true*. For delay behaviour,  $b \rightarrow p$  can delay according to  $p$  as long as the boolean expression  $b$  evaluates to *true*. While  $b$  evaluates to *false*,  $b \rightarrow p$  can perform any delay.
2. The sequential composition operator ( $;$ ). A process  $p; q$  behaves as  $p$  followed by  $q$ .
3. The alternative composition operator ( $\square$ ). A process  $p \square q$  represents a non-deterministic choice between  $p$  and  $q$  if they can proceed.
4. The repetition operator ( $*$ ). A process  $*p$  behaves as  $p$  infinitely many times.
5. The parallel composition operator ( $\parallel$ ). A process  $p \parallel q$  executes  $p$  and  $q$  concurrently in an interleaved fashion, i.e. the actions of  $p$  and  $q$  are executed in arbitrary

order. If one of the processes can execute a *send* action and the other one can execute a *receive* action on the same channel then  $p \parallel q$  executes the communication action on this channel.

6. The scope operator ( $\llbracket \_ \rrbracket$ ). A process  $\llbracket s \mid p \rrbracket$  behaves as  $p$  in a local state  $s$ . The state  $s$  is used to define local variables and channels visible only to the process  $p$ . It is recursively defined as the empty state or as  $dcl, s'$  where  $s'$  is a state and  $dcl$  is a variable declaration ( $x : type[= val]$ ) or a channel declaration ( $h : ?type$  for receiving,  $h : !type$  for sending, and  $h : -type$  for both).
7. The encapsulation operator ( $\partial_{\mathcal{A}}$ ). A process  $\partial_{\mathcal{A}}(p)$  disables all actions of  $p$  that occur in the set  $\mathcal{A}$ . Typically this operator is used to enforce that send and receive actions synchronise.
8. The urgent communication operator ( $v_{\mathcal{H}}$ ). Send and receive actions in a process  $v_{\mathcal{H}}(p)$  via channels from set  $\mathcal{H}$  can only delay when no communication with a corresponding receive or send action on the same channel is possible.

**Process definitions.** The language  $\chi_t$  provides the possibility to define processes. We do not give a syntax definition here but rather an example:

$$P(c : ? \text{nat}, b : \text{bool}) = \llbracket x : \text{nat} \mid b \rightarrow c ? x \rrbracket$$

The process  $P$  has two arguments, a channel  $c$  that can transport natural numbers and a boolean variable  $b$ . It has only one local variable,  $x$ . The process can now be instantiated at the initialisation line as for example  $P(m, y > 7)$  (using channel  $m$  and natural number  $y$ , both declared at the initialisation line).

### 3 The language $\mu\text{CRL}$

Basically,  $\mu\text{CRL}$  is based on the process algebra ACP [4], extended with equational abstract data types [27]. In order to intertwine processes with data, actions and recursion variables can be parametrised with data types. Moreover, a conditional construct (if-then-else) can be used to have data elements influence the course of a process, and *alternative quantification* (also called *choice quantification*) is added to sum over possibly infinite data domains.

The language comes with a toolset [8] that can build a state space from a specification and store it in the `.aut` format, one of the input formats of the model checker CADP [15]. Next to that, in order to strive for precision in proofs, an important research area is to use theorem provers such as PVS [30] to help in finding and checking derivations in  $\mu\text{CRL}$ . A large number of distributed systems have been

verified in  $\mu\text{CRL}$ , often with the help of a proof checker or theorem prover [16, 21].

We will give a short overview of the language necessary for understanding this paper. For a complete reference, see [17].

**Data types.** Initially there are no data types known in a  $\mu\text{CRL}$  specification. Therefore each specification should start by defining the necessary data types and the functions that work on them. In fact, it is mandatory to define the boolean type in each specification, since the conditional construct works with boolean expressions. One can virtually define any data type. In an example at the end of this paper we use a data type for the natural numbers.

**Actions.** In  $\mu\text{CRL}$  one can declare actions in the `act` section of a specification. These actions may have zero, one or several data parameters. One can also allow processes  $P$  and  $Q$  to communicate in the parallel process  $P \parallel Q$ . To do this it is possible to define which actions are able to synchronise with each other in the `comm` section of a specification.

Finally the process deadlock ( $\delta$ ), which cannot terminate successfully, and the internal action  $\tau$  are predefined.

**Operators.** There are eight operators in  $\mu\text{CRL}$ . We omit the renaming operator and the abstraction operator since we do not use them in this paper. We present the other six with an informal semantics.

1. The alternative composition operator ( $+$ ). A process  $p+q$  proceeds (non-deterministically) as  $p$  or  $q$  (if they can proceed).
2. The sum operator ( $\sum_{d \in D} X(d)$ ), with  $X(d)$  a mapping from the data type  $D$  to processes, behaves as  $X(d_1) + X(d_2) + \dots$ , i.e., as the possibly infinite choice between  $X(d)$  for any data term  $d$  taken from  $D$ . This operator is used to describe a process that is reading some input over a data type [28].
3. The sequential composition operator ( $\cdot$ ). A process  $p \cdot q$  proceeds as  $p$  followed by  $q$ .
4. The process expression  $p \triangleleft b \triangleright q$  where  $p$  and  $q$  are processes, and  $b$  is a data term of data type `Bool`, behaves as  $p$  if  $b$  is equal to `T` (true) and behaves as  $q$  if  $b$  is equal to `F` (false). This operator is called the conditional operator.
5. The parallel composition operator ( $\parallel$ ). A process  $p \parallel q$  executes  $p$  and  $q$  concurrently in an interleaved fashion, i.e. the actions of  $p$  and  $q$  are executed in arbitrary order. For all actions  $a$  and  $b$  which can communicate with each other: If one process can execute  $a$  and the

other one can execute  $b$  then  $p$  and  $q$  can communicate ( $p \parallel q$  executes the communication action).

6. The encapsulation operator ( $\partial_H$ ). A process  $\partial_H(p)$  disables all actions of  $p$  that occur in the set  $H \subseteq \text{Act}$ . Typically this operator is used to enforce that certain actions synchronise.

**Process definitions.** The heart of a  $\mu\text{CRL}$  specification is the `proc` section, where the behaviour of the system is declared. This section consists of recursion equations of the following form, for  $n \geq 0$ :

$$\text{proc } X(x_1 : s_1, \dots, x_n : s_n) = t$$

Here  $X$  is the process name, the  $x_i$  are variables, not clashing with the name of a function symbol of arity zero nor with a parameterless process or action name, and the  $s_i$  are data type names, expressing that the data parameters  $x_i$  are of type  $s_i$ . Moreover,  $t$  is a process term possibly containing occurrences of expressions  $Y(d_1, \dots, d_m)$ , where  $Y$  is a process name and the  $d_i$  are data terms that may contain occurrences of the variables  $x_1, \dots, x_n$ . In this rule,  $X(x_1, \dots, x_n)$  is declared to have the same (potential) behaviour as the process expression  $t$  [17].

The initial state of the specification is declared in a separate initial declaration `init` section, which is of the form

$$\text{init } X(d_1, \dots, d_n)$$

Here  $d_1, \dots, d_n$  represent the initial values of the parameters  $x_1, \dots, x_n$ . In  $\mu\text{CRL}$  specifications the `init` section is used to instantiate the data parameters of a process declaration, meaning that the  $d_i$  are data terms that do not contain variables. The `init` section may be omitted, in which case the initial behaviour of the system is left unspecified.

**The time model.** Delaying for a certain amount of time is impossible in  $\mu\text{CRL}$  at first glance. This is because  $\mu\text{CRL}$  does not work with time. A later extension of  $\mu\text{CRL}$  to *timed*  $\mu\text{CRL}$  [20] introduced the notion of time. However, at present creating a timed  $\mu\text{CRL}$  specification is not very practical since the  $\mu\text{CRL}$  toolset can only parse timed  $\mu\text{CRL}$  code and cannot generate a state space from it.

There is another way however to simulate some notion of discrete time. In this paper we use a method based on the one from [9]. In short it works like this: first we define two actions: `tick` and `tick2`. The `tick` action represents the end of a time slice and the beginning of a new one. In order to share this notion of time all running processes need to synchronise their `tick` actions. If at least one of these processes is busy and therefore unable to perform a `tick` the `tick` action will not take place. This synchronisation aspect is essential if one wants to use global timing. Note

that, using this technique, we get discrete time in  $\mu\text{CRL}$ , since we represent a time period as a number of time units.

In most cases when using time in a model the modeller would like to give normal actions priority over `tick` actions. In order to realise this  $\chi$  has implicit urgent communication, but in  $\mu\text{CRL}$  an operator for this does not exist. We can however get similar results by using the `tick2` action and post-processing the system after linearisation (more on the latter in section 4.9).

The differences between `tick2` and `tick` are:

- The action `tick` is used for translating delays, while `tick2` is used to make an action delayable (which means adding a `tick2` self-loop as an alternative to this action);
- A `tick` action can synchronise with any number of `tick` or `tick2` actions, but a `tick2` action cannot synchronise with only `tick2` actions (at least one `tick` action is needed for going from one time unit to the next).

Now, several delayable processes can delay together if there is a `tick` action enabled in at least one process.

## 4 The translation scheme

### 4.1 Linear process equations

In this paper we use a slightly extended version of the linear process equation (LPE) definition as stated in [7]. An LPE is a one-line process declaration that consists of atomic actions, summations, sequential compositions and conditionals. In particular, an LPE does not contain any parallel operators, encapsulations or hidings. In essence an LPE is a vector of data parameters together with a list of summands consisting of a condition, action and effect triple, describing when an action may happen and what its effect is on the vector. This format resembles I/O automata [29], extended finite state machines [25], Unity processes [13] and STGA [26]. An LPE is of the following form:

$$\begin{aligned} X(d : D) = & \\ & \sum_{i \in I} \sum_{e_i \in D_i} a_i(f_i(d, e_i)).X(g_i(d, e_i)) \triangleleft h_i(d, e_i) \triangleright \delta + \\ & \sum_{i \in I'} \sum_{e_i \in D_i'} a_i'(f_i'(d, e_i)).\checkmark(g_i'(d, e_i)) \triangleleft h_i'(d, e_i) \triangleright \delta \end{aligned}$$

where  $I, I'$  are finite index sets,  $D, D_i, D_i', D_{a_i}$  and  $D_{a_i'}$  are data types,  $a_i, a_i' \in \mathbf{Act} \cup \{\tau\}$ ,  $a_i : D_{a_i}, a_i' : D_{a_i'}$ ,  $f_i : D \times D_i \rightarrow D_{a_i}, f_i' : D \times D_i' \rightarrow D_{a_i'}$ ,  $g_i : D \times D_i \rightarrow D$ ,  $g_i' : D \times D_i' \rightarrow D$ ,  $h_i : D \times D_i \rightarrow \mathbf{Bool}$  and  $h_i' : D \times D_i' \rightarrow \mathbf{Bool}$ .

Here the different states of the process are represented by the data parameter  $d : D$ . Type  $D$  may be a Cartesian product of  $n$  data types. Besides that the data parameter  $e_i$  (either of type  $D_i$  or  $D_i'$ ) can influence the parameter of action  $a_i$  (or  $a_i'$ ), the condition  $h_i$  (or  $h_i'$ ) and the resulting state

$g_i$  (or  $g_i'$ ), thereby giving LPEs a more general form. The data parameter  $e_i$  is typically used to let a read action range over a data domain.

The extension to the definition from [7] is the usage of  $\checkmark$ . Using the original definition, process  $X$  would terminate after executing an action  $a_i'$  after which it would be impossible to state anything about the end state. Here however  $\checkmark(g_i'(d, e_i))$  should be read as “the process enters state  $g_i'(d, e_i)$  after which it successfully terminates” (the process has reached an end state).

In general, when translating a  $\chi_t$  process to an LPE the variables  $s_i$  in the scope operator of  $\chi_t$  should be translated to parameters of the LPE (should become part of the data parameter  $d : D$ ). Channels in  $\chi_t$  that a process works with are mentioned as parameters in the  $\chi_t$  specification of that process, but these should not be included in the LPE.

In both  $\chi_t$  and  $\mu\text{CRL}$  one can use data types. We could go into detail concerning how an element of a data type in  $\chi_t$  can be translated to an element of a data type in  $\mu\text{CRL}$ , but we will not do so here. It suffices to say that this kind of translation is rather trivial; for virtually any data type in  $\chi_t$  one can define a corresponding abstract data type in  $\mu\text{CRL}$ .

### 4.2 Initialisation

The parallel composition operator and the encapsulation operator are here placed in one section, since both of them are used in a particular way within a  $\mu\text{CRL}$  specification. More specific, in the  $\mu\text{CRL}$  toolset, these operators are only allowed to be used in the initialisation line. What follows are guidelines to translate these operators and which assumptions are made during the remainder of this chapter.

In general the initialisation line of a  $\chi_t$  specification looks like this:

$$\langle x_k : ty_k, h_m : -th_m \mid p_1() \parallel \dots \parallel p_n() \rangle$$

Here  $x_k : ty_k, h_m : -th_m$  is an abbreviation for  $x_1 : ty_1, \dots, x_k : ty_k, h_1 : -th_1, \dots, h_m : -th_m$ . This declares discrete variables  $x_1, \dots, x_k$  of types  $ty_1, \dots, ty_k$ , respectively, and channels  $h_1, \dots, h_m$  that communicate information of types  $th_1, \dots, th_m$ , respectively. Furthermore  $p_1(), \dots, p_n()$  are processes.

In this extended abstract we will not discuss how to translate the variables that are declared at the initialisation line; they are global variables and cannot be translated in a straightforward fashion. These variables are discussed in more detail in the full version of this article [31].

However, each channel which is declared at the initialisation line should be translated to a send action, a corresponding receive action, and a corresponding communication action, defined in the `act` section of the  $\mu\text{CRL}$  specification. Then a rule should be added to the `comm` section, allowing the send action and the receive action to communicate.

The usage of the parallel composition operator at initialisation can be translated in a straightforward fashion. In the

initialisation line we see the parallel composition operator being used to specify which processes make up the system (in other words, which processes run in parallel from the start). The initialisation line of a  $\mu\text{CRL}$  specification looks like this:

$$\text{init } \partial_{\mathbb{H}}(p_1() \parallel \dots \parallel p_n())$$

Here we can see that the parallel composition operator is used in the same way.

In  $\chi_t$  the parallel composition operator can also be used inside processes. In such a case the result is really a process consisting of subprocesses running in parallel. Since this usage of the parallel composition operator is not allowed in  $\mu\text{CRL}$ , we want to avoid these constructions in  $\chi_t$ . Soon, however, there will be a new  $\mu\text{CRL}$  lineariser available, which does allow nested parallelism. Then this will no longer be a restriction.

The encapsulation operator of  $\chi_t$  ( $\partial_{\mathcal{A}}$ ) is implicitly used at initialisation. It can be translated by using the encapsulation operator of  $\mu\text{CRL}$  ( $\partial_{\mathbb{H}}$ ), making the set  $\mathbb{H}$  equal to the set  $\mathcal{A}$ .

For the remainder of this chapter we assume that a  $\chi_t$  specification ready for translation has the previously displayed initialisation line with processes  $p_1(), \dots, p_n()$  not containing the parallel composition operator, the encapsulation operator and the urgent communication operator (more on the latter in section 4.9).

### 4.3 Atomic processes

**The multi-assignment process.** In  $\mu\text{CRL}$  assignments take place by using recursion or calling a new process in which the new value of the changed variable is given as a parameter. Therefore, process  $x_n := e_n$  can be translated to  $\mathbf{X}(d : D) = \tau \cdot \checkmark(d[e_1/x_1, \dots, e_n/x_n])$ , in which  $\checkmark(d[e_1/x_1, \dots, e_n/x_n])$  means that you end up in a state where  $e_1, \dots, e_n$  have been substituted for  $x_1, \dots, x_n$  respectively, while the other variables in the state remain unchanged.

**The skip process.** The process skip performs the internal action  $\tau$ . This can be translated into an LPE by using the  $\tau$  action. The translation then becomes  $\mathbf{X}(d : D) = \tau \cdot \checkmark(d)$ .

**The send process.** In  $\mu\text{CRL}$  channels are not available as a type, like they are in  $\chi_t$ . Instead one can define actions and synchronise these with each other. Traditionally, sending a command like e.g. *test* can be done by using an action **stest** (the **s** stands for *send*). This command can be received by another process with the action **rtest**, where the **r** means *receive*. The actions **stest** and **rtest** must be defined in the specification, together with a communication rule, saying that a send over the test 'channel' together with a receive over this 'channel' leads to a communication (an action called **ctest**). It is important that when describing

the initial situation one encapsulates the send and receive actions in order to force communication between the two.

Taking into account that a send process  $h!e$  should be delayable, this has to be translated to  $\mathbf{X}(d : D) = \text{sh}(e) \cdot \checkmark(d) + \text{tick2} \cdot \mathbf{X}(d)$  with **sh** being the action of sending something over the channel  $h$ .

**The receive process.** As mentioned in the previous paragraph  $\mu\text{CRL}$  does not work with channels, but one can define send and receive actions and force them to communicate. Receive actions traditionally begin with the letter **r**, which would mean in this case that the receive action would be defined as **rh**.

The process  $h?x$  translates to

$$\mathbf{X}(d : D) = \sum_{y:Ty} (\text{rh}(y) \cdot \checkmark(d[y/x])) + \text{tick2} \cdot \mathbf{X}(d)$$

with  $Ty$  being the type of both  $x$  and  $y$ .

**The delay process.** The translation of the delay process  $\Delta t$  is highly dependent on the time model used in  $\mu\text{CRL}$ . Therefore the reader should be aware of this time model as described in the time model paragraph of section 3. Note that while  $\chi_t$  uses continuous time, this time model only considers discrete time. Therefore, only the "discrete time part" of  $\chi_t$  can be translated.

If we restrict the possible values of  $t$  in  $\Delta t$  to the natural numbers then we can translate the delay to the following LPE, where  $\mathbf{t}$  and  $\mathbf{t}_0$  are both translations of  $t$ :

$$\mathbf{X}(\mathbf{t} : \text{Nat}, \mathbf{t}_0 : \text{Nat}, d : D) = \text{tick} \cdot \mathbf{X}(\mathbf{t} - 1, \mathbf{t}_0, d) \triangleleft \mathbf{t} > 0 \triangleright \delta + \tau \cdot \checkmark(\mathbf{t}_0, \mathbf{t}_0, d) \triangleleft \mathbf{t} = 0 \triangleright \delta$$

We cannot set the initial value of  $\mathbf{t}$  and  $\mathbf{t}_0$  in process  $\mathbf{X}$ . We have to do that in the initialisation line. We introduce a set  $V$ , which contains all initial values of process parameters. When writing the initialisation line, we obtain the parameter values from this set. For now we set the initial value of both  $\mathbf{t}$  and  $\mathbf{t}_0$  in  $V$  to the value of  $t$ . Counter  $\mathbf{t}_0$  is used to be able to reset counter  $\mathbf{t}$  to its initial value. It is important that this is done after executing process  $\mathbf{X}$  to allow the possibility to repeat execution of  $\mathbf{X}$  by means of the repetition operator (see section 4.7). Finally note that both  $\mathbf{t}$  and  $\mathbf{t}_0$  are underlined in the parameter list. This has been done to indicate that these parameters are specially marked. For the use of this see section 4.7.

### 4.4 Guard operator

When discussing the translation of  $\chi_t$  operators in the upcoming sections, two LPEs  $P$  and  $Q$  will be used. These

processes have the following form:

$$\begin{aligned}
P(d : D) = & \sum_{i \in I} \sum_{e_i \in D_i} a_i(f_{p_i}(d, e_i)).P(g_{p_i}(d, e_i)) \triangleleft hp_i(d, e_i) \triangleright \delta + \\
& \sum_{i \in I'} \sum_{e_i \in D_i'} a_i'(f_{p_i'}(d, e_i)).\checkmark(g_{p_i'}(d, e_i)) \\
& \triangleleft hp_i'(d, e_i) \triangleright \delta \\
Q(d' : D') = & \sum_{j \in J} \sum_{e_j \in D_j} a_j(f_{q_j}(d', e_j)).Q(g_{q_j}(d', e_j)) \triangleleft hq_j(d', e_j) \triangleright \delta + \\
& \sum_{j \in J'} \sum_{e_j \in D_j'} a_j'(f_{q_j'}(d', e_j)).\checkmark(g_{q_j'}(d', e_j)) \\
& \triangleleft hq_j'(d', e_j) \triangleright \delta
\end{aligned}$$

We avoid name clashes of variables in  $d$  and  $d'$  when it is necessary to combine the two LPEs.

In the upcoming sections we use a function  $\mathcal{T} : \chi \rightarrow \mu\text{CRL}$  which gets a  $\chi_t$  process as input and returns an LPE as output. This function provides a translation by induction on the structure of a  $\chi_t$  process.

Consider a process  $b \rightarrow p$  with  $b$  being a boolean expression. Say LPE  $P = \mathcal{T}(p)$  and  $b$  is a translation of the guard  $b$ . Finally we say that the finite index set  $I = I_n \cup I_t$ , with  $I_n \cap I_t = \emptyset$ ,  $I_n$  being a set of indexes of all actions in  $P$  which are not `tick` or `tick2` actions (i.e. 'normal' actions) and  $I_t$  being a set of indexes of all actions in  $P$  which are either `tick` or `tick2` actions. For  $I'$  we do not have to do a similar thing since  $I_t'$  will always be empty. A process never terminates after executing a `tick` or `tick2` action; after a `tick` action there is always eventually a normal action (see the translation of the delay process) and `tick2` actions only occur in self-loops.

Now  $\mathcal{T}(b \rightarrow p)$  is defined as follows:

$$\begin{aligned}
X(n : \text{Nat}, d : D) = & \sum_{i \in I_n} \sum_{e_i \in D_i} a_i(f_{p_i}(d, e_i)).X(1, g_{p_i}(d, e_i)) \\
& \triangleleft hp_i(d, e_i) \wedge (n = 1 \vee b) \triangleright \delta + \\
& \sum_{i \in I_t} \sum_{e_i \in D_i} a_i(f_{p_i}(d, e_i)).X(n, g_{p_i}(d, e_i)) \\
& \triangleleft hp_i(d, e_i) \wedge (n = 1 \vee b) \triangleright \delta + \\
& \sum_{i \in I'} \sum_{e_i \in D_i'} a_i'(f_{p_i'}(d, e_i)).\checkmark(0, g_{p_i'}(d, e_i)) \\
& \triangleleft hp_i'(d, e_i) \wedge (n = 1 \vee b) \triangleright \delta + \\
& \text{tick2}.X(n, d) \triangleleft n = 0 \wedge \neg b \triangleright \delta
\end{aligned}$$

Basically the following things have been done to combine the boolean expression  $b$  and the LPE  $P$ :

1. A counter  $n$  has been introduced. It has type  $\text{Nat}$  but in practise  $n \in \{0, 1\}$ . This counter is initially 0 (we set this in the set  $V$ ) and is used here to regulate that only initially the value of  $b$  is important.
2. Notice the difference between the first and the second line: Instead of being set to 1 the counter  $n$  is un-

changed. This is very important when  $n = 0$ , since this means that the value of the boolean expression  $b$  remains important in the next time unit.

3. In the third line  $n$  is reset to 0. Why this is done can be read in section 4.7 on the repetition operator.
4. In the fourth line it is expressed that if  $b$  does not hold and no 'normal' action has been executed yet ( $n = 0$ ) this process can delay one time unit without changing the current state.
5. In all lines the guard has been expanded with equations concerning  $n$  and  $b$  to express that one may only start executing 'normal' actions if  $b$  holds.

## 4.5 Sequential composition operator

Assume we have the  $\chi_t$  process  $p; q$  with the LPE  $P = \mathcal{T}(p)$  and the LPE  $Q = \mathcal{T}(q)$ . Now we define  $\mathcal{T}(p; q)$  as follows:

$$\begin{aligned}
X(n : \text{Nat}, d : D, d' : D') = & \sum_{i \in I} \sum_{e_i \in D_i} a_i(f_{p_i}(d, e_i)).X(0, g_{p_i}(d, e_i), d') \\
& \triangleleft hp_i(d, e_i) \wedge n = 0 \triangleright \delta + \\
& \sum_{i \in I'} \sum_{e_i \in D_i'} a_i'(f_{p_i'}(d, e_i)).X(1, g_{p_i'}(d, e_i), d') \\
& \triangleleft hp_i'(d, e_i) \wedge n = 0 \triangleright \delta + \\
& \sum_{j \in J} \sum_{e_j \in D_j} a_j(f_{q_j}(d', e_j)).X(1, d, g_{q_j}(d', e_j)) \\
& \triangleleft hq_j(d', e_j) \wedge n = 1 \triangleright \delta + \\
& \sum_{j \in J'} \sum_{e_j \in D_j'} a_j'(f_{q_j'}(d', e_j)).\checkmark(0, d, g_{q_j'}(d', e_j)) \\
& \triangleleft hq_j'(d', e_j) \wedge n = 1 \triangleright \delta
\end{aligned}$$

A counter  $n$  has been introduced to regulate the order of execution. Initially this counter has value 0, thereby enabling the execution of the actions originally from the LPE  $P$ . At those points where  $P$  terminates successfully,  $n$  is set to 1, disabling the execution of actions from  $P$  and enabling the execution of actions from  $Q$ .

## 4.6 Alternative composition operator

In this section we give a translation of the  $\chi_t$  process  $p \parallel q$ , which chooses non-deterministically between the processes  $p$  and  $q$ . At first glance providing a translation for this does not seem to be more difficult than providing one for the sequential composition. This, however, turns out to be untrue, due to the time mechanism of  $\chi_t$ ; if both alternatives  $p$  and  $q$  can delay, then they delay *together*. If only one alternative can delay and furthermore no actions can be executed at all then there is a deadlock. Finally, time does not make a choice, meaning that if the process delays before a

choice for one of the alternatives has been made the process still has to make this choice after that delay.

Say we have the  $\chi_t$  process  $p \parallel q$  with the LPE  $P = \mathcal{T}(p)$  and the LPE  $Q = \mathcal{T}(q)$ . Furthermore we say that the finite index set  $I = I_n \cup I_t$  (similar to section 4.4). Finally we say that  $I_t = I_{t1} \cup I_{t2}$ , with  $I_{t1} \cap I_{t2} = \emptyset$ ,  $I_{t1}$  being a set of indexes of all occurrences of the `tick` action in  $P$  and  $I_{t2}$  being a set of indexes of all occurrences of the `tick2` action in  $P$ . In a similar way we define  $J = J_n \cup J_t$  and  $J_t = J_{t1} \cup J_{t2}$ . Now we define  $\mathcal{T}(p \parallel q)$  as follows:

$$\begin{aligned}
X(n : \text{Nat}, d : D, d' : D') = & \\
& \sum_{i \in I_n} \sum_{e_i \in D_i} a_i(f_{p_i}(d, e_i)).X(1, g_{p_i}(d, e_i), d') \\
& \quad \langle \text{hp}_i(d, e_i) \wedge (n = 0 \vee n = 1) \triangleright \delta \ + \\
& \sum_{i \in I_t} \sum_{e_i \in D_i} a_i(f_{p_i}(d, e_i)).X(n, g_{p_i}(d, e_i), d') \\
& \quad \langle \text{hp}_i(d, e_i) \wedge n = 1 \triangleright \delta \ + \\
& \sum_{i \in I'} \sum_{e_i \in D_i'} a_i(f_{p_i'}(d, e_i)).\checkmark(0, g_{p_i'}(d, e_i), d') \\
& \quad \langle \text{hp}_i'(d, e_i) \wedge (n = 0 \vee n = 1) \triangleright \delta \ + \\
& \sum_{j \in J_n} \sum_{e_j \in D_j} a_j(f_{q_j}(d', e_j)).X(2, d, g_{q_j}(d', e_j)) \\
& \quad \langle \text{hq}_j(d', e_j) \wedge (n = 0 \vee n = 2) \triangleright \delta \ + \\
& \sum_{j \in J_t} \sum_{e_j \in D_j} a_j(f_{q_j}(d', e_j)).X(n, d, g_{q_j}(d', e_j)) \\
& \quad \langle \text{hq}_j(d', e_j) \wedge n = 2 \triangleright \delta \ + \\
& \sum_{j \in J'} \sum_{e_j \in D_j'} a_j(f_{q_j'}(d', e_j)).\checkmark(0, d, g_{q_j'}(d', e_j)) \\
& \quad \langle \text{hq}_j'(d', e_j) \wedge (n = 0 \vee n = 2) \triangleright \delta \ + \\
& \sum_{i \in I_{t1}} \sum_{j \in J_{t2}} \sum_{e_i \in D_i} \sum_{e_j \in D_j} \text{tick}.X(n, g_{p_i}(d, e_i), g_{q_j}(d', e_j)) \\
& \quad \langle \text{hp}_i(d, e_i) \wedge \text{hq}_j(d', e_j) \wedge n = 0 \triangleright \delta \ + \\
& \sum_{i \in I_{t2}} \sum_{j \in J_{t1}} \sum_{e_i \in D_i} \sum_{e_j \in D_j} \text{tick}.X(n, g_{p_i}(d, e_i), g_{q_j}(d', e_j)) \\
& \quad \langle \text{hp}_i(d, e_i) \wedge \text{hq}_j(d', e_j) \wedge n = 0 \triangleright \delta \ + \\
& \sum_{i \in I_{t1}} \sum_{j \in J_{t1}} \sum_{e_i \in D_i} \sum_{e_j \in D_j} \text{tick}.X(n, g_{p_i}(d, e_i), g_{q_j}(d', e_j)) \\
& \quad \langle \text{hp}_i(d, e_i) \wedge \text{hq}_j(d', e_j) \wedge n = 0 \triangleright \delta \ + \\
& \sum_{i \in I_{t2}} \sum_{j \in J_{t2}} \sum_{e_i \in D_i} \sum_{e_j \in D_j} \text{tick2}.X(n, d, d') \\
& \quad \langle \text{hp}_i(d, e_i) \wedge \text{hq}_j(d', e_j) \wedge n = 0 \triangleright \delta
\end{aligned}$$

Basically the following things have been done to combine the LPEs  $P$  and  $Q$ :

1. A counter  $n$  has been introduced. It has type  $\text{Nat}$  but in practise  $n \in \{0, 1, 2\}$ .
2. In the first line we find the 'normal' actions that originally are not at the end of process  $P$  ( $P$  does not terminate after performing one of these actions). Since  $n$  initially equals 0, some of these actions can be performed in the beginning of executing  $X$  (where  $\text{hp}_i(d, e_i)$  holds).

3. In the second line we find all occurrences of `tick` and `tick2` in LPE  $P$ . It is very important to note that the usage of  $n$  in the guard (only considering  $n = 1$ ) leads to guards which are always false in cases where `tick` and `tick2` actions are enabled in the beginning of executing  $P$ . This is because initially  $n$  does not equal 1, but 0 and after that, when  $n$  does equal 1,  $\text{hp}_i(d, e_i)$  does not hold. This results in `tick` and `tick2` occurrences at the beginning of  $P$  (in terms of execution order) being effectively removed from process  $X$ .
4. In the third line we find the 'normal' actions as we did in the first line, only after executing these actions  $P$  originally terminates. As we see here,  $X$  terminates as well.
5. In the fourth, fifth and sixth line we find situations similar to the first, second and third line respectively, only now they concern actions from process  $Q$ .
6. In line seven we combine all occurrences of `tick` in process  $P$  with all occurrences of `tick2` in process  $Q$ . Together these form `tick` occurrences in  $X$ , where the new state is defined by using the two functions  $g_{p_i}$  and  $g_{q_j}$  and the guard is the conjunction of the guards of the occurrences being combined together with the expression  $n = 0$ . This last expression  $n = 0$  effectively makes all guards equal to  $F$ , except in those cases where both the `tick` and the `tick2` occurrence are at the beginning (execution-wise) of  $P$  and  $Q$ , respectively. The reason for this is similar to the one given for the second line.
7. In the same way as is done in the previous line, the remaining lines combine `tick2` occurrences in  $P$  with `tick` occurrences in  $Q$ , `tick` occurrences in  $P$  and  $Q$  and `tick2` occurrences in  $P$  and  $Q$ , respectively.

So, in line two and five the `tick` and `tick2` occurrences from the beginning of  $P$  and  $Q$  are practically removed, only to appear in a combined form in lines seven, eight and nine. This reflects what happens in the  $\chi_t$  process  $p \parallel q$ , where  $p$  and  $q$  delay together if they can both delay and no delay will happen if one of them cannot.

## 4.7 Repetition operator

When executing the  $\chi_t$  process  $*p$ , the process  $p$  gets executed in sequence infinitely often. This construction needs to be translated using recursion.

Say we have a  $\chi_t$  process  $*p$  where the LPE  $P = \mathcal{T}(p)$ . Now we define  $\mathcal{T}(*p)$  as follows:

$$\begin{aligned}
X(d : D) = & \\
& \sum_{i \in I} \sum_{e_i \in D_i} a_i(f_{p_i}(d, e_i)).X(g_{p_i}(d, e_i)) \langle \text{hp}_i(d, e_i) \triangleright \delta \ + \\
& \sum_{i \in I'} \sum_{e_i \in D_i'} a_i(f_{p_i'}(d, e_i)).X(\underline{g_{p_i'}}(d, e_i)) \langle \text{hp}_i'(d, e_i) \triangleright \delta
\end{aligned}$$

In the LPE the check-mark ( $\checkmark$ ) has been replaced by  $\mathbb{X}$ , resulting in executing  $\mathbb{X}$  from the beginning again every time  $\mathbb{X}$  has executed the final action in the LPE. When repeating the execution the LPE automatically begins with the first action, which is assured by the translation of  $\mathbb{p}$  (in all translations of the operators, counters get their initial value back at termination). However, note that the new state, when the process starts repeating, is underlined. This is done to indicate that all marked parameters  $\mathbf{t}$  in  $\mathbf{d}$  (see section 4.3) are assigned the values of their accompanying parameters  $\mathbf{t}_0$ . The reason for this is that should process  $P$  contain an alternative composition, it is not always the case that all timers are reset to their initial values upon termination.

#### 4.8 Scope operator

The process algebra  $\mu\text{CRL}$  does not have a scope operator, but the functionality of this can be found implicitly in the algebra. Note that in  $\chi_t$  the state  $s$  is used to define local programming variables or local channels. So in a way, a state is a tuple consisting of variables and channels. In  $\mu\text{CRL}$  the programming variables of a process can be found in its parameter  $\mathbf{d}$  (initial values can be found in the initialisation line) while the channels are defined as send and receive actions globally. In other words,  $s$  is captured in  $\mu\text{CRL}$  by recursion parameters and global action and communication definitions. Therefore, there is no direct translation of the scope operator needed. As the  $\chi_t$  local channels are translated to  $\mu\text{CRL}$  global actions, some extra work is needed to make these actions seem local. More on this in the full article [31].

#### 4.9 Urgent communication operator

In  $\mu\text{CRL}$  there is no urgent communication operator. We can however still get similar results using  $\mu\text{CRL}$ , which will be explained next.

In order to translate the urgent communication operator we first need to linearise the translation. This means that translating urgent communication can only be done once all other translations are ready. This reflects nicely the fact that in  $\chi_t$ , urgent communication is added to a system once it is completed.

Say we have a  $\chi_t$  process  $v_{\mathcal{H}}(p)$  where the LPE  $P = \mathcal{T}(p)$  and  $\mathcal{H}$  contains all channels used by  $p$ . Now we define  $\mathcal{T}(v_{\mathcal{H}}(p))$  as follows:

$$\begin{aligned} \mathbb{X}(\mathbf{d} : \mathbf{D}) = & \sum_{i \in \mathcal{I}_n} \sum_{\mathbf{e}_i \in \mathcal{D}_i} \mathbf{a}_i(\mathbf{f}_i(\mathbf{d}, \mathbf{e}_i)) . \mathbb{X}(\mathbf{g}_i(\mathbf{d}, \mathbf{e}_i)) \triangleleft \mathbf{h}_i(\mathbf{d}, \mathbf{e}_i) \triangleright \delta + \\ & \sum_{i \in \mathcal{I}_t} \sum_{\mathbf{e}_i \in \mathcal{D}_i} \mathbf{a}_i(\mathbf{f}_i(\mathbf{d}, \mathbf{e}_i)) . \mathbb{X}(\mathbf{g}_i(\mathbf{d}, \mathbf{e}_i)) \\ & \triangleleft \mathbf{h}_i(\mathbf{d}, \mathbf{e}_i) \wedge \neg \bigvee_{j \in \mathcal{I}_n} \mathbf{h}_j(\mathbf{d}, \mathbf{e}_i) \triangleright \delta \end{aligned}$$

Note that the finite index set  $\mathcal{I} = \mathcal{I}_n \cup \mathcal{I}_t$ , as used before.

## 5 An example: the system $PQ$

Concluding we look at an example  $\chi_t$  specification in order to illustrate how translation works on a concrete case study. For a bigger example and a demonstration how to verify properties of a  $\mu\text{CRL}$  model, look at [11]. The example we use here is a system consisting of the processes  $P$  and  $Q$ . The  $\chi_t$  specification of the system is the following:

$$\begin{aligned} P(a : !\text{bool}) = & \llbracket i : \text{nat} = 2 \\ & | *(i \geq 1 \rightarrow \Delta 3.0; a ! \text{false}; i := i + 1 \\ & | i \geq 2 \rightarrow a ! \text{true}; i := i - 1) \rrbracket \\ Q(a : ?\text{bool}) = & \llbracket b : \text{bool} \\ & | *( \Delta 2.0; a ? b) \rrbracket \\ \langle a : -\text{bool} | P(a) \parallel Q(a) \rangle \end{aligned}$$

Next we translate these two processes to LPEs. After that we would linearise that using the  $\mu\text{CRL}$  toolset and then introduce urgent communication. We do not do that here however, due to space limitations. This is done in the full version of this article [31]. Here, the example is only used to show how a  $\chi_t$  system can be translated process by process.

We start by noting that the only data types used are the ones for the natural numbers and the booleans. For  $\mu\text{CRL}$  this means we will use the data types `Nat` and `Bool`. Since these are standard we do not display their definitions here.

Now we detect the channels used and define appropriate actions for them in  $\mu\text{CRL}$ . In the  $\chi_t$  specification we see the channel  $a$ . For this we define the actions `sa`, `ra` and `ca`, all three having a boolean value as parameter. Here `sa` stands for sending a value, `ra` is used for receiving a value and `ca` represents the communication between the two. We define that `sa` can communicate with `ra`, forming action `ca`.

Concerning process  $P$ , we know how to translate the individual actions. Using a single counter, for readability purposes, we can place the actions in the right structure (following the translation scheme we would end up with a list of counters, but here we use only one counter, which can range over the set of natural numbers). Furthermore we see two guards placed in an alternative composition. Finally this construction is subject to the repetition operator. Translating all this (and simplifying it by removing those actions which will never be executed due to their guards never being true) we get:

$$\begin{aligned} \text{proc } P(\mathbf{n} : \text{Nat}, \mathbf{t} : \text{Nat}, \mathbf{t}_0 : \text{Nat}, \mathbf{i} : \text{Nat}) = & \\ & \text{tick}.P(\mathbf{n}, \mathbf{t} - 1, \mathbf{t}_0, \mathbf{i}) \triangleleft \mathbf{t} > 0 \wedge \mathbf{n} = 0 \wedge \mathbf{i} \geq 1 \triangleright \delta + \\ & \tau.P(1, \mathbf{t}_0, \mathbf{t}_0, \mathbf{i}) \triangleleft \mathbf{t} = 0 \wedge \mathbf{n} = 0 \wedge \mathbf{i} \geq 1 \triangleright \delta + \\ & \text{sa}(\mathbf{F}).P(2, \mathbf{t}, \mathbf{t}_0, \mathbf{i}) \triangleleft \mathbf{n} = 1 \triangleright \delta + \\ & \text{tick2}.P(\mathbf{n}, \mathbf{t}, \mathbf{t}_0, \mathbf{i}) \triangleleft \mathbf{n} = 1 \triangleright \delta + \\ & \tau.P(0, \mathbf{t}_0, \mathbf{t}_0, \mathbf{i} + 1) \triangleleft \mathbf{n} = 2 \triangleright \delta + \\ & \text{sa}(\mathbf{T}).P(3, \mathbf{t}, \mathbf{t}_0, \mathbf{i}) \triangleleft \mathbf{n} = 0 \wedge \mathbf{i} \geq 2 \triangleright \delta + \\ & \tau.P(0, \mathbf{t}_0, \mathbf{t}_0, \mathbf{i} - 1) \triangleleft \mathbf{n} = 3 \triangleright \delta + \\ & \text{tick2}.P(\mathbf{n}, \mathbf{t}, \mathbf{t}_0, \mathbf{i}) \triangleleft \mathbf{i} < 1 \wedge \mathbf{n} = 0 \triangleright \delta \end{aligned}$$

Finally we translate process  $Q$ . This is a very small process which is translated to the following LPE:

$$\begin{aligned} \text{proc } Q(n : \text{Nat}, t : \text{Nat}, t_0 : \text{Nat}, b : \text{Bool}) = \\ \text{tick}.Q(n, t - 1, t_0, b) \triangleleft t > 0 \wedge n = 0 \triangleright \delta + \\ \tau.P(1, t_0, t_0, b) \triangleleft t = 0 \wedge n = 0 \triangleright \delta + \\ \sum_{b_0 : \text{Bool}} \text{ra}(b_0).Q(0, t_0, t_0, b_0) \triangleleft n = 1 \triangleright \delta + \\ \text{tick2}.Q(n, t, t_0, b) \triangleleft n = 1 \triangleright \delta \end{aligned}$$

The initialisation line makes the translation complete:

$$\text{init } \partial_{\{\text{sa}, \text{ra}, \text{tick2}\}}(P(0, 3, 3, 2) \mid \{\text{tick2}\} \mid Q(0, 2, 2, F))$$

Here,  $\mid \{\text{tick2}\} \mid$  is a special operator which functions as the normal parallel composition operator, but also forces a correct usage of time progression ( $\text{tick}$  and  $\text{tick2}$  actions have to synchronise with each other). Notice that we encapsulate  $\text{tick2}$  eventually, which results in the fact that the synchronisation of a number of  $\text{tick2}$  actions (without any  $\text{tick}$  action) will not lead to an action in the system.

Now that this translation is finished, we could linearise and post-process it to introduce urgent communication.

## References

- [1] D. v. Beek, A. v. d. Ham, and J. Rooda. Modelling and Control of Process Industry Batch Production Systems. In *Proc. IFAC'02*, 2002.
- [2] D. v. Beek, K. Man, M. Reniers, J. Rooda, and R. Schiffelers. Syntax and Consistent Equation Semantics of Hybrid Chi. CS-Report 04-37, Eindhoven University of Technology, 2004.
- [3] D. v. Beek, K. Man, M. Reniers, J. Rooda, and R. Schiffelers. Syntax and Semantics of Timed Chi. CS-Report 05-09, Eindhoven University of Technology, 2005.
- [4] J. Bergstra and J. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, 1984.
- [5] J. Bergstra and J. Klop. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, 1998.
- [6] M. Bernardo, W. Cleaveland, S. Sims, and W. Stewart. TwoTowers: A Tool Integrating Functional and Performance Analysis of Concurrent Systems. In *Proc. FORTE/PSTV'98*, pp. 457–467. Kluwer, 1998.
- [7] M. Bezem and J. Groote. Invariants in Process Algebra with Data. In *Proc. CONCUR '94*, vol. 836 of LNCS, pp. 401–416, 1994.
- [8] S. Blom, W. Fokkink, J. Groote, I. v. Langevelde, B. Lisser, and J. v. d. Pol.  $\mu\text{CRL}$ : A Toolset for Analysing Algebraic Specifications. In *Proc. CAV 2001*, vol. 2102 of LNCS, pp. 250–254, 2001.
- [9] S. Blom, N. Ioustinova, and N. Sidorova. Timed verification with  $\mu\text{CRL}$ . In *Proc. PSI 2003*, vol. 2890 of LNCS, pp. 178–192, 2003.
- [10] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.
- [11] E. Bortnik, N. Trčka, A. Wijs, S. Luttik, J. v. d. Mortel-Fronczak, J. Baeten, W. Fokkink, and J. Rooda. Analyzing a  $\chi$  Model of a Turntable System using SPIN, CADP and UPPAAL. *Journal of Logic Programming*, To appear, 2005.
- [12] V. Bos and J. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer Integrated Manufacturing*, 17:185–198, 2001.
- [13] K. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.
- [14] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. CAV 2002*, vol. 2404 of LNCS, pp. 359–364, 2002.
- [15] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP - a protocol validation and verification toolbox. In *Proc. CAV'96*, vol. 1102 of LNCS, pp. 437–440, 1996.
- [16] W. Fokkink, J. Groote, J. Pang, B. Badban, and J. v. d. Pol. Verifying a Sliding Window Protocol in  $\mu\text{CRL}$ . In *Proc. AMAST 2004*, vol. 3116 of LNCS, pp. 148–163, 2004.
- [17] W. Fokkink, J. Groote, and M. Reniers. Modelling Distributed Systems. Unpublished manuscript, 2002.
- [18] W. Fokkink, N. Ioustinova, E. Kesseler, J. v. d. Pol, Y. Usenko, and Y. Yushtein. Refinement and verification applied to an in-flight data acquisition unit. In *Proc. CONCUR 2002*, vol. 2421 of LNCS, pp. 1–23, 2002.
- [19] H. Garavel and H. Hermanns. On Combining Functional Verification and Performance Evaluation Using CADP. In *Proc. FME 2002*, vol. 2391 of LNCS, pp. 410–429, 2002.
- [20] J. Groote. The Syntax and Semantics of timed  $\mu\text{CRL}$ . Technical Report SEN-R9709, CWI, 1997.
- [21] J. Groote, F. Monin, and J. v. d. Pol. Checking verifications of protocols and distributed systems by computer. In *Proc. CONCUR'98*, vol. 1466 of LNCS, pp. 629–655, 1998.
- [22] J. Groote, J. Pang, and A. Wouters. Analysis of a distributed system for lifting trucks. *Journal of Logic and Algebraic Programming*, 55(1-2):21–56, 2003.
- [23] H. Hermanns and J.-P. Katoen. Performance Evaluation := (Process Algebra + Model Checking)  $\times$  Markov Chains. In *Proc. CONCUR 2001*, vol. 2154 of LNCS, pp. 59–81, 2001.
- [24] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [25] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, June 1994.
- [26] H. Lin. Symbolic transition graph with assignment. In V. Sassone, editor, *Proc. CONCUR'96*, number 1119 in LNCS, pp. 50–65. Springer-Verlag, 1996.
- [27] J. Loeckx, H.-D. Ehrlich, and M. Wolf. *Specification of Abstract Data Types*. Wiley-Teubner, Chichester, Stuttgart, 1996.
- [28] S. Luttik. *Choice Quantification in Process Algebra*. PhD thesis, University of Amsterdam, 2002.
- [29] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [30] S. Owre, J. Rushby, and N. Shankar. PVS: a Prototype Verification System. In *Proc. CADE'92*, vol. 607 of LNCS, pp. 748–752, 1992.
- [31] A. Wijs and W. Fokkink. From  $\chi_t$  to  $\mu\text{CRL}$ : Combining Performance and Functional Analysis. Technical Report SEN-R0420, CWI, Amsterdam, 2004.