Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 215 (2008) 191–208
www.elsevier.com/locate/entcs

Process Algebra with Local Communication

Muck van Weerdenburg[1]

Eindhoven University of Technology
Department of Mathematics and Computer Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract

In process algebras like μCRL and ACP communication is defined globally. In the context of component-based architectures one wishes to define subcomponents of a system separately, including communication within that subcomponent. We define a process algebra with an operator for local communication that facilitates component-based architectures. Besides being compositional, this language is aimed to be a more practical language (with respect to closely related languages) and also allows for straightforward modelling of synchronous as well as asynchronous behaviour.

Keywords: process algebra, local communication, true concurrency, compositionality, synchrony

1 Introduction

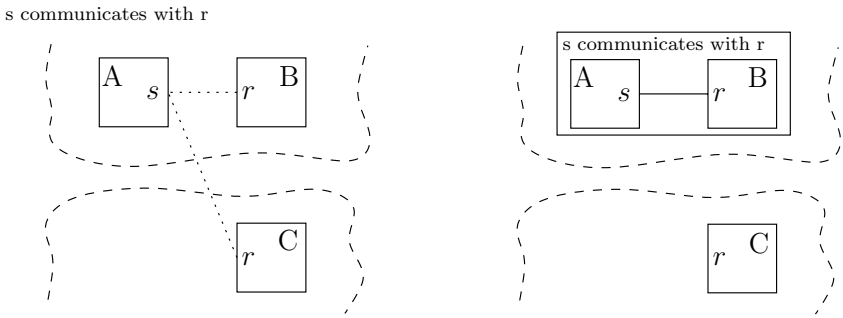
In modelling systems, component-based architectures are a natural way of separating different parts (and subparts) of a system and specifying how these parts relate to each other by means of, for example, communication. Especially in the case of larger systems (i.e. real-life systems and complex protocols), component-based modelling is essential to avoid losing overview of the model. We introduce a new process algebra called *LoCo* which aim it is to be a practical language supporting such hierarchical modelling. Besides the asynchronous behaviour seen in most languages, this algebra also supports the modelling of synchronous behaviour. This allows for the modelling of, for example, electronic circuits (from which one wants to abstract away from the relatively small delays) or easy multiway communications.

[1] E-Mail: M.J.van.Weerdenburg@tue.nl

doi:10.1016/j.entcs.2008.06.028

The main difference between *LoCo* and most other process algebras is the fact that it has a local communication mechanism and multiactions. Local communication means that one can specify communication between components precisely where it is relevant, whereas global communication means that one has to specify the communication for the whole system in one place. Multiactions are basically multisets of actions that occur at the same time (without communicating). With these concepts we also get a straightforward way to model synchronous behaviour.

To illustrate the differences between global and local communication we consider Figure 1. Here a system with two of components, A and B, that desire to communicate via actions s and r is depicted. Now say that in another part of the system there is a components C that, for some purpose, also uses an action r (possibly because C is actually just B but in a different context). In Figure 1(a) it is illustrated that it is not possible to simply define these communications on a global level. There is no way for component A to avoid communicating with C instead of B. The only way to avoid such mistakes is to rename some of the actions (e.g. r in A to r_A etc.). With local communication one can simply specify that a given communication is only meant for certain parts of the system. This is illustrated on the right in Figure 1(b).



(a) Global communication (b) Local communication

Fig. 1. Global vs. local communication

Primarily, the language *LoCo* was developed as a basis for mCRL2[2] [13,15], successor of μCRL [14], where an important motivation was to be able to straightforwardly express Petri nets [17] in process algebra. In Petri nets the firing of a transition consist of taking tokens from some places and, at the same time, putting tokens in others. This was the initial reason to

[2] mCRL2 is basically *LoCo* with a slightly different syntax, timing and a (more or less) fixed data algebra.

introduce multiactions, which in turn resulted in the addition of local instead of global communication. After adding this local communication it became clear that it made the language a truly compositional process algebra. This led us to reversing the thought process; local communication is introduced to make the language compositional and it is this local communication that leads to the introduction of multiactions (as is explained below).

It is due to this origin that *LoCo* seems to be more closely related to other attempts to link Petri nets and process algebra than to process algebras developed for component-based modelling. Also the fact that the latter are often inspired by CCS [20] or the π-calculus [22] plays a role. These languages, such as CaSE [23], IP-calculus [10] and PiLar [11], have local communication, but it is restricted to actions (a) communicating with their counterparts ($\overline{a}$) only. Or, as in PADL [6], they use strict synchronisation on actions. In such cases, communication (or synchronisation to a single action) is tightly linked to the parallel operators and therefore does not require multiactions. It does, however, restrict the freedom in modelling and complicates constructions as multiway communication. Other process algebras with local communication or synchronisation, such as LOTOS [16] and the Interworkings language [19], also have this strong connection between communication and parallelism.

In *LoCo* we have separated the concepts of parallelism and communication and linked them by means of multiactions. This also gives a more natural intuition to the operators. Parallel composition now means just putting two components in parallel; the execution of actions in one component is independent of the other component. Only by applying a special operator for communication one creates links between components. To make this possible, multiactions are needed. The parallel composition results in multiactions as a consequence of components executing actions at the same time. Creating a link between two (or more) components is done by applying the local communication operator. This way actions within the multiactions result in a communication. It is also much easier to model multiway communication in this way, in contrast to algebras where communication and parallelism are as one. In these languages one has to specify communication per pair of components (and pair of pair of components etc.) in such a way that the result is in fact a multiway communication. That is, if it is even possible to do so only by means of communication.

This separation of parallelism and communication is also seen in $\mathrm{ACP}_{\mathrm{ec}}$ [3] and the algebra from [2], which describe an ACP [4] approach to combining process algebra and Petri nets. However, in [3] the communication used by the local communication operator is still defined globally and thus limits compositionality. Because of this one cannot use a single component multiple times in

the same system (in different contexts) without renaming its actions. On the other hand, [2] uses a renaming operator to apply local communication, but in a setting with data this is not really a preferable choice. It is more intuitive for renaming to disregard data and for communication to require that data parameters (of actions) are equivalent.

The Petri Box Calculus [8,7,9] gives a more Petri-net-based approach to combining both formalisms. Although the communication is done in the CCS style, multiactions are used here. Because of these multiactions multiway communication becomes feasible, but it remains cumbersome due to the type of communication. A component needs to know that it is going to participate in a multiway communication or one has to introduce a special component that takes care of it. The same holds for SCCS [21] and MEIJE [1], which add synchrony to CCS.

What is not encountered in any of the above algebras with multiactions is a restriction operator that only allows certain actions. Only in [2] one might consider it present in the blocking (or encapsulation) operator (the inverse of restriction), but blocking on its own is not sufficient in practice. The reason to add the restriction operator is that it can often be the case that one only wishes to allow a small number of multiactions of a process with a much greater total amount of different multiactions. Take for example the parallel composition of n actions, which results in $2^n - 1$ different multiactions. If one only wants to allow the multiaction in which each action is present, blocking would require one to specify $2^n - 2$ multiactions (instead of just one with our restriction operator).

In Section 2 we introduce the syntax of *LoCo* for which we give a semantics in Section 3. An axiomatisation for *LoCo* is given in Section 4 to facilitate algebraic reasoning. As abstraction and data often play an important role in systems, we discuss these in Sections 5 and 6, respectively. Some examples are given in Section 7 to illustrate *LoCo* by modelling a Petri net and a simple compositional system.

2 Syntax

We describe the elements of our algebra informally. A detailed definition of the syntax can be found in [28] (without data) and [27] (with data).

The basic elements of processes are multiactions. Multiactions are bags (or multisets) of actions, from the set of actions $\mathcal{N}_A$, that execute together. We write a multiaction of actions a, b and c as $\langle a, b, c\rangle$ (or $\langle b, a, c\rangle$ as order has no meaning in bags). Often we write multiactions that consist of only one action without brackets (i.e. a instead of $\langle a\rangle$). We can combine such multiactions

with the common operators $\cdot$ and $+$ to form a sequence of multiactions or a nondeterministic choice between multiactions, respectively. The special case of the empty multiaction $\langle\rangle$ is called a silent step, which we often write as τ. To denote inaction or deadlock we write δ.

For parallel composition we have the merge $\|$ which interleaves and/or synchronises multiactions. A communication operator Γ allows explicit specification of (two or more) actions that communicate with each other (e.g. $\Gamma_{\{a|b\to c\}}(\langle a,b,b\rangle+\langle a\rangle)$, meaning that a and b communicate to c in $\langle a,b,b\rangle+\langle a\rangle$, is equivalent to $\langle c,b\rangle+\langle a\rangle$).

To limit the behaviour of a process, it has been common to define which actions are *not* allowed. However, the number of multiactions we want to prohibit can increase exponentially with the number of parallel processes; putting n actions in parallel results in 2^n-1 different multiactions. Therefore we added a restriction operator ∇ that specifies precisely which multiactions *are* allowed, by a set V of action sequences (e.g. $V=\{a\}$ or $V=\{b|c|c,d,c|e\}$). If one wishes that in the parallel composition of a,b and c action a does not execute synchronised with another action and b and c must synchronise, one can write $\nabla_{\{a,b|c\}}(a\parallel b\parallel c)$, which behaves as $a\cdot\langle b,c\rangle+\langle b,c\rangle\cdot a$.

The blocking operator ∂_H (commonly referred to as encapsulation operator) prohibits *actions* in its set parameter H from executing (e.g. $\partial_{\{a\}}(a+b\cdot\langle a,c\rangle)$, which behaves as $b\cdot\delta$), the hiding operator τ_I makes actions in I invisible (e.g. $\tau_{\{a\}}(\langle a,b\rangle)$ becomes $\langle b\rangle$) and the renaming operator ρ renames actions (e.g. $\rho_{\{a\to b\}}(a)$ becomes b). Finally we have process references with which we can write definitions such as $P=a\cdot P$, which denotes the process that can do infinitely many a actions.

Table 1, where $\mathcal{V}=\{a_1|\ldots|a_n \ : \ a_1,\ldots,a_n\in\mathcal{N}_A\}$ the set of possible action-name combinations, $\mathcal{C}=\{a_1|\ldots|a_n\to b \ : \ a_1,\ldots,a_n,b\in\mathcal{N}_A\}$ the set of possible communications and $\mathcal{R}=\{a\to b \ : \ a,b\in\mathcal{N}_A\}$ the set of all renamings, contains a summary of the above.

Note that this syntax allows one to write sets (R) that can contain elements with the same left hand side (e.g. $\{a\to b,a\to c\}$). This should not be possible as the meaning of these sets are meant to be functions. Therefore we put the restriction on this syntax that in the sets described by R no left hand side of an element may be the same as the left hand side of another.

For the set C a similar restriction holds. Specifically, left hand sides must be disjoint, meaning that $\{a|b\to c,d|b\to e\}$ is not allowed as b occurs in both left hand sides. Otherwise communication applied to $\langle a,b,d\rangle$ could result in either $\langle c,d\rangle$ or $\langle a,e\rangle$, which we consider to needlessly complicate the communication (such behaviour can better be explicitly modelled by the user, in our opinion).

Table 1
LoCo Syntax

$a, b, c, \ldots$	Single actions (from $\mathcal{N}_A$, also $\langle a\rangle$, $\langle b\rangle$, ...)
$\langle a\rangle, \langle b, c, d, b\rangle, \langle\rangle, \ldots$	Multiactions (containing single actions)
δ	Deadlock/inaction
τ	Silent step (also $\langle\rangle$)
$\_ + \_$	Alternative composition
$\_ \cdot \_$	Sequential composition
$\_ \parallel \_$	Merge/Parallel composition
$\_ \lfloor\!\lfloor \_$	Left merge
$\_ \mid \_$	Synchronisation operator
$P, Q, \ldots$	Process references
$\nabla_V(\_)$	Restriction operator ($V \subseteq \mathcal{V}$)
$\Gamma_C(\_)$	Communication operator ($C \subseteq \mathcal{C}$)
$\partial_H(\_)$	Blocking operator (Encapsulation, $H \subseteq \mathcal{N}_A$)
$\tau_I(\_)$	Hiding operator ($I \subseteq \mathcal{N}_A$)
$\rho_R(\_)$	Renaming operator ($R \subseteq \mathcal{R}$)
$P = \_$	Process definition (with P a process reference)

Some additional notation is used in this document for ease of reading. Instead of writing the sequence of terms $t_1, t_2, \ldots, t_n$ (e.g. the actions in a multiaction) we often write $\boldsymbol{t}$. We also write α, β etc. instead of the multiactions like $\langle \boldsymbol{a}\rangle$.

Now that we have introduced our syntax, we will have a look at some examples of *LoCo* processes. Process $M = (\mathit{coin} \parallel \mathit{button}) \cdot \mathit{product} \cdot M$ models a simple vendor machine that waits for a user to insert a coin and press a button (in any order) and then gives a product. This process has the same behaviour as $M' = (\mathit{coin} \cdot \mathit{button} + \mathit{button} \cdot \mathit{coin} + \langle \mathit{coin}, \mathit{button}\rangle) \cdot \mathit{product} \cdot M'$, where the third alternative indicates that it is possible to insert a coin at the same time as pressing the button. Another example is $\nabla_{\{a,b\}}(\tau_{\{s_{ab}\}}(\Gamma_{\{s_a|s_b\to s_{ab}\}}(A \parallel B)))$, with $A = a \cdot s_a$ and $B = s_b \cdot b$, which models two separate processes A and B that have to synchronise such that a happens before b. This process has the same behaviour as $a \cdot b$ (in branching bisimulation semantics [25]).

3 Operational Semantics

For the definition of the semantics of *LoCo* we need some auxiliary notation (for precise definitions, see [28]). First of all, we will use the set of all multiactions $\mathbb{A} = \{\langle \boldsymbol{a}\rangle \ : \ \boldsymbol{a} \in \boldsymbol{A}\}$ and we use $|$ to combine multiactions (i.e. $\langle \boldsymbol{a}\rangle|\langle \boldsymbol{b}\rangle$

is just $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$). To be able to reason about terms of our language, we introduce some sets and notations. The set V_P, with elements $x, y, \ldots$, consists of process variables. For the set of *LoCo* terms (described by) T_P we have elements $t, u, \ldots$ and process-closed terms $p, q, \ldots$ in T_{pc} (terms that do not have any process variables or references in them).

In Table 2 we give the operational semantics of *LoCo*. We use the standard transition relations $\longrightarrow$ and $\longrightarrow \checkmark$, and assume we have a set E containing process definitions. Note that the semantics of the operators ∇_V, Γ_C, ∂_H, τ_I and ρ_R is given separately.

Table 2
LoCo Semantics

$$\frac{}{a \xrightarrow{\langle a \rangle} \checkmark} \qquad \frac{}{\alpha \xrightarrow{\alpha} \checkmark} \qquad \frac{}{\tau \xrightarrow{\langle\rangle} \checkmark}$$

$$\frac{t \xrightarrow{\alpha} \checkmark}{\begin{array}{l} t + u \xrightarrow{\alpha} \checkmark \\ u + t \xrightarrow{\alpha} \checkmark \end{array}} \qquad \frac{t \xrightarrow{\alpha} t'}{\begin{array}{l} t + u \xrightarrow{\alpha} t' \\ u + t \xrightarrow{\alpha} t' \end{array}} \qquad \frac{t \xrightarrow{\alpha} \checkmark}{t \cdot u \xrightarrow{\alpha} u} \qquad \frac{t \xrightarrow{\alpha} t'}{t \cdot u \xrightarrow{\alpha} t' \cdot u}$$

$$\frac{t \xrightarrow{\alpha} \checkmark}{\begin{array}{l} t \parallel u \xrightarrow{\alpha} u \\ u \parallel t \xrightarrow{\alpha} u \end{array}} \qquad \frac{t \xrightarrow{\alpha} t'}{\begin{array}{l} t \parallel u \xrightarrow{\alpha} t' \parallel u \\ u \parallel t \xrightarrow{\alpha} u \parallel t' \end{array}} \qquad \frac{t \xrightarrow{\alpha} \checkmark}{t \between u \xrightarrow{\alpha} u} \qquad \frac{t \xrightarrow{\alpha} t'}{t \between u \xrightarrow{\alpha} t' \parallel u}$$

$$\frac{t \xrightarrow{\alpha} \checkmark, u \xrightarrow{\beta} \checkmark}{t \parallel u \xrightarrow{\alpha|\beta} \checkmark} \qquad \frac{t \xrightarrow{\alpha} \checkmark, u \xrightarrow{\beta} u'}{\begin{array}{l} t \parallel u \xrightarrow{\alpha|\beta} u' \\ u \parallel t \xrightarrow{\alpha|\beta} u' \end{array}} \qquad \frac{t \xrightarrow{\alpha} t', u \xrightarrow{\beta} u'}{t \parallel u \xrightarrow{\alpha|\beta} t' \parallel u'}$$

$$\frac{t \xrightarrow{\alpha} \checkmark, u \xrightarrow{\beta} \checkmark}{t|u \xrightarrow{\alpha|\beta} \checkmark} \qquad \frac{t \xrightarrow{\alpha} \checkmark, u \xrightarrow{\beta} u'}{\begin{array}{l} t|u \xrightarrow{\alpha|\beta} u' \\ u|t \xrightarrow{\alpha|\beta} u' \end{array}} \qquad \frac{t \xrightarrow{\alpha} t', u \xrightarrow{\beta} u'}{t|u \xrightarrow{\alpha|\beta} t' \parallel u'}$$

$$\frac{t \xrightarrow{\alpha} \checkmark}{P \xrightarrow{\alpha} \checkmark}\ P = t \in E \qquad \frac{t \xrightarrow{\alpha} t'}{P \xrightarrow{\alpha} t'}\ P = t \in E$$

The definition of the operators ∇_V, Γ_C, ∂_H, τ_I and ρ_R in the semantics requires some functions that perform the needed transformations or checks on multiactions. To start with the blocking operator ∂_H, we need to test whether or not an action in H occurs in a multiaction. We do this by converting such a multiaction α to a set $\alpha_{\{\}}$ (i.e. if α is $\langle a, b, c, c \rangle$, then $\alpha_{\{\}}$ will be $\{a, b, c\}$) and then taking the intersection of $\alpha_{\{\}}$ and H, which gives all actions that occur in both α and H.

Restriction operator ∇_V needs to check whether or not a given multiaction occurs in its set V (or is τ or $\langle\rangle$, which is always allowed). Because the set

V does not contain multiactions but action sequences of the form $a|a|b$, we convert V to $V_{\langle\rangle}$, which is defined by $\{\langle a_1, \ldots, a_n\rangle \ : \ a_1|\ldots|a_n \in V\}$.

To apply renaming, defined by R in ρ_R, to a multiaction α, we write $R \bullet \alpha$. For example, if we apply $R = \{a \to b, b \to a\}$ to $\langle a, b, b, c\rangle$, we get $R \bullet \langle a, b, b, c\rangle = \langle R(a), R(b), R(b), R(c)\rangle = \langle b, a, a, c\rangle$.

With hiding τ_I, we need to remove all actions in I from multiactions. We introduce a special function θ_I for this purpose. Thus, $\theta_{\{a,b\}}(\langle d, a, b, a, c\rangle)$ would result in $\langle d, c\rangle$ and $\theta_{\{a\}}(\langle a, a\rangle)$ in $\langle\rangle$ (or τ).

For the communication operator we need a somewhat more complex definition. We introduce a communication function γ_C that takes a multiaction and finds all occurrences of left hand sides in C and replaces those occurrences with the corresponding right hand side. To be somewhat more precise (note that we implicitly use the commutativity of bags in this definition):

$$\begin{array}{lcll} \gamma_C(\langle a_1, \ldots, a_n\rangle|\alpha) & = & \langle c\rangle|\gamma_C(\alpha) & \text{if } \ a_1|\ldots|a_n \to c \ \in \ C \\ \gamma_C(\alpha) & = & \alpha & \text{if there is no such } a_1, \ldots, a_n \end{array}$$

So, if C is $\{a|b \ \to \ a, c|c|d \ \to \ b\}$, then $\gamma_C(\langle a, a, a, b, b, c, c, d\rangle)$ results in $\langle a, a, a, b\rangle$. Note that the extra condition on C (discussed in Section 2) is required to make γ a true function. That is, $\gamma_C(\alpha)$ is a unique multiaction, which follows from the fact that if an action can participate in two possible communications, then these have to be equivalent due to the restriction on C (e.g. a can communicate with the first or the second b of $\langle a, b, b\rangle$, but either way the effect is the same).

With these auxiliary functions the semantics of *LoCo* is completed with the rules from Table 3.

Table 3
LoCo Semantics (continued)

$$\frac{t \xrightarrow{\alpha} \checkmark}{\nabla_V(t) \xrightarrow{\alpha} \checkmark}\ \alpha \in V_{\langle\rangle} \cup \{\langle\rangle\} \qquad \frac{t \xrightarrow{\alpha} t'}{\nabla_V(t) \xrightarrow{\alpha} \nabla_V(t')}\ \alpha \in V_{\langle\rangle} \cup \{\langle\rangle\}$$

$$\frac{t \xrightarrow{\alpha} \checkmark}{\Gamma_C(t) \xrightarrow{\gamma_C(\alpha)} \checkmark} \qquad \frac{t \xrightarrow{\alpha} t'}{\Gamma_C(t) \xrightarrow{\gamma_C(\alpha)} \Gamma_C(t')}$$

$$\frac{t \xrightarrow{\alpha} \checkmark}{\partial_H(t) \xrightarrow{\alpha} \checkmark}\ \alpha_{\{\}} \cap H = \emptyset \qquad \frac{t \xrightarrow{\alpha} t'}{\partial_H(t) \xrightarrow{\alpha} \partial_H(t')}\ \alpha_{\{\}} \cap H = \emptyset$$

$$\frac{t \xrightarrow{\alpha} \checkmark}{\tau_I(t) \xrightarrow{\theta_I(\alpha)} \checkmark} \qquad \frac{t \xrightarrow{\alpha} t'}{\tau_I(t) \xrightarrow{\theta_I(\alpha)} \tau_I(t')}$$

$$\frac{t \xrightarrow{\alpha} \checkmark}{\rho_R(t) \xrightarrow{R \bullet \alpha} \checkmark} \qquad \frac{t \xrightarrow{\alpha} t'}{\rho_R(t) \xrightarrow{R \bullet \alpha} \rho_R(t')}$$

To be able to compare and calculate with processes, we need to know when two processes are equal (i.e. have the same behaviour). We therefore use the default notion of (strong) bisimilarity [20,24] and write $LoCo \vDash p \underline{\leftrightarrow} q$ (or just $p \underline{\leftrightarrow} q$) to denote that p and q are (strongly) bisimilar. And as the rules in Table 2 and Table 3 are in the *path* format [5], we have that bisimulation $\underline{\leftrightarrow}$ is a congruence with respect to all operators.

4 Axioms

We introduce the axiomatisation in Table 4 for the semantics given in the previous section. The axioms allow for more straightforward reasoning in certain cases. It also shows that *LoCo* has a reasonably elegant algebraic structure similar to those of other process algebras.

Table 4
LoCo Axioms

$MA1$	$a \doteq \langle a \rangle$	VD	$\nabla_V(\delta) \doteq \delta$
$MA2$	$\tau \doteq \langle\rangle$	$V1$	$\nabla_V(\alpha) \doteq \alpha$ if $\alpha \in V_{\langle\rangle} \cup \{\langle\rangle\}$
$MA3$	$\langle \boldsymbol{a}, \boldsymbol{b} \rangle \doteq \langle \boldsymbol{b}, \boldsymbol{a} \rangle$	$V2$	$\nabla_V(\alpha) \doteq \delta$ if $\alpha \notin V_{\langle\rangle} \cup \{\langle\rangle\}$
		$V3$	$\nabla_V(x+y) \doteq \nabla_V(x) + \nabla_V(y)$
$A1$	$x + y \doteq y + x$	$V4$	$\nabla_V(x \cdot y) \doteq \nabla_V(x) \cdot \nabla_V(y)$
$A2$	$x + (y + z) \doteq (x + y) + z$		
$A3$	$x + x \doteq x$	DD	$\partial_H(\delta) \doteq \delta$
$A4$	$(x + y) \cdot z \doteq x \cdot z + y \cdot z$	$D1$	$\partial_H(\alpha) \doteq \alpha$ if $\alpha_{\{\}} \cap H = \emptyset$
$A5$	$(x \cdot y) \cdot z \doteq x \cdot (y \cdot z)$	$D2$	$\partial_H(\alpha) \doteq \delta$ if $\alpha_{\{\}} \cap H \neq \emptyset$
$A6$	$x + \delta \doteq x$	$D3$	$\partial_H(x+y) \doteq \partial_H(x) + \partial_H(y)$
$A7$	$\delta \cdot x \doteq \delta$	$D4$	$\partial_H(x \cdot y) \doteq \partial_H(x) \cdot \partial_H(y)$
$CM1$	$x \parallel y \doteq x \between y + y \between x + x \mid y$	TID	$\tau_I(\delta) \doteq \delta$
$CM2$	$\alpha_\delta \between x \doteq \alpha_\delta \cdot x$	$TI1$	$\tau_I(\alpha) \doteq \theta_I(\alpha)$
$CM3$	$\alpha_\delta \cdot x \between y \doteq \alpha_\delta \cdot (x \parallel y)$	$TI3$	$\tau_I(x+y) \doteq \tau_I(x) + \tau_I(y)$
$CM4$	$(x + y) \between z \doteq x \between z + y \between z$	$TI4$	$\tau_I(x \cdot y) \doteq \tau_I(x) \cdot \tau_I(y)$
$CM5$	$(\alpha_\delta \cdot x) \mid \beta_\delta \doteq (\alpha_\delta \mid \beta_\delta) \cdot x$		
$CM6$	$\alpha_\delta \mid (\beta_\delta \cdot x) \doteq (\alpha_\delta \mid \beta_\delta) \cdot x$	RD	$\rho_R(\delta) \doteq \delta$
$CM7$	$(\alpha_\delta \cdot x) \mid (\beta_\delta \cdot y) \doteq (\alpha_\delta \mid \beta_\delta) \cdot (x \parallel y)$	$R1$	$\rho_R(\alpha) \doteq R \bullet \alpha$
$CM8$	$(x + y) \mid z \doteq x \mid z + y \mid z$	$R3$	$\rho_R(x+y) \doteq \rho_R(x) + \rho_R(y)$
$CM9$	$x \mid (y + z) \doteq x \mid y + x \mid z$	$R4$	$\rho_R(x \cdot y) \doteq \rho_R(x) \cdot \rho_R(y)$
$CM10$	$\langle \boldsymbol{a} \rangle \mid \langle \boldsymbol{b} \rangle \doteq \langle \boldsymbol{a}, \boldsymbol{b} \rangle$		
		GD	$\Gamma_C(\delta) \doteq \delta$
$CD1$	$\delta \mid \alpha_\delta \doteq \delta$	$G1$	$\Gamma_C(\alpha) \doteq \gamma_C(\alpha)$
$CD2$	$\alpha_\delta \mid \delta \doteq \delta$	$G3$	$\Gamma_C(x+y) \doteq \Gamma_C(x) + \Gamma_C(y)$
		$G4$	$\Gamma_C(x \cdot y) \doteq \Gamma_C(x) \cdot \Gamma_C(y)$

With $a, b \in \mathcal{N}_A$, $\alpha, \beta \in \mathbb{A}$, $\alpha_\delta, \beta_\delta \in \mathbb{A} \cup \{\delta\}$ and $x, y \in V_P$.

If we can derive q from p with the axioms (in the ordinary equational sense), we write $LoCo \vdash p \doteq q$ (or just $p \doteq q$). And if we do so, we obviously want the same to hold for bisimulation, which is stated in the following theorem.

Theorem 4.1 *Let $p, q \in T_{pc}$. The axiomatisation of LoCo is sound with respect to strong bisimulation (i.e. $LoCo \vdash p \doteq q \Rightarrow LoCo \vDash p \underline{\leftrightarrow} q$).*

Proof. This proof is very straightforward and therefore not given here. However, it can be found in [28].

The other way around is also a desired property.

Theorem 4.2 *Let $p, q \in T_{pc}$. The axiomatisation of LoCo is complete with respect to strong bisimulation (i.e. $LoCo \vDash p \leftrightarrows q \Rightarrow LoCo \vdash p \doteq q$).*

Proof. The full proof can be found in [28].

Because we also wish to use recursive processes, we will (at least) need some extension to the axioms given in Table 4. We extend $\doteq$ with the following rule:

$$\frac{P = t \in E}{P \doteq t}$$

From the rules for process definitions in Section 3 the soundness of this rule clearly follows.

5 Abstraction

If we want τ (or empty multiset $\langle\rangle$) to be a "real" *silent step*, we want to be able to remove τ where its presence cannot be determined. Due to the nature of multiactions, it is already the case that if τ is synchronised with (i.e. executed at the same time as) some action α, the τ "disappears" (applying axiom $MA2$ to $\tau|\alpha$ gives $\langle\rangle|\alpha$ and with $CM10$ we get just α). A multiaction is not just a multiset of actions, but a multiset of *observable* actions. There is always the possibility that *unobservable* actions happen at the same time. Note that this behaviour of the synchronisation operator is not to be confused with the behaviour of the communication operator in, for example, ACP, which would deadlock (because τ cannot communicate). Also note that this behaviour is the same as seen in, for example, the Petri Box Calculus and similar to that in SCCS (apart from the different interpretation of this multiaction identity).

However, this is not the only place where we wish to hide these silent steps. In these cases, strong bisimulation is no longer suitable and we will use (rooted) branching bisimulation [25,26]. For this form of equivalence, we also need a matching (i.e. sound and complete) axiomatisation. Fortunately, the axioms given before are still sound, but to make the axiomatisation complete again (i.e. to have axioms that reflect the behaviour of τ) it is sufficient to add the following two axioms, as in [12].

$$T1 \quad x \cdot \tau \doteq x$$

$$T2 \quad x \cdot (\tau \cdot (y + z) + y) \doteq x \cdot (y + z)$$

Theorem 5.1 *The axiomatisation of LoCo is sound with respect to (rooted) branching bisimulation.*

Proof. In [28] one can find the soundness proofs of the axioms $T1$ and $T2$. The other axioms of *LoCo*, which have already been proven to be sound with respect to $\leftrightarrow$, do not need to be proven again, as $\leftrightarrow \subset \leftrightarrow_{rb}$ holds.

Theorem 5.2 *The axiomatisation of LoCo is complete with respect to rooted branching bisimulation.*

Proof. The proof is similar to that in [12].

6 Data

In many systems, data plays an essential role and is therefore a necessity for any practical language. We add data by adding parameters to actions. So, instead of actions a, b etc. we now also have actions like $a(1)$, $b(\mathit{true}, [c_0, c_1(1, 2), c_0])$ and $b(4, \mathit{error})$. The precise data expressions that are allowed as parameters are defined by a data algebra $\mathcal{A}$. All we need to know about it is that it contains a number of data types, one of which is the boolean type (with the default constants ($\boldsymbol{t}$, $\boldsymbol{f}$) and operators). Detailed requirements are given in [27], as well as a more detailed definition of syntax and semantics with data.

We also need the summation $\sum_{d:D} p$, the conditional operator $b \rightarrow p$ and data parameters for process references. The behaviour of $\sum_{d:D} p$ is the same as the alternative composition of all $p[e/d]$ (i.e. p with every unbounded occurrence of d replaced by e), for each e in data type D. Conditional $b \rightarrow p$ behaves as p or deadlock if the boolean condition b is equivalent to $\boldsymbol{t}$ or $\boldsymbol{f}$, respectively. With process references with data we can write definitions such as $P(n : \mathbb{N}) = \sum_{m:\mathbb{N}} m < n \rightarrow a(m)$, meaning that, for example, process reference $P(5)$ is equivalent to $a(0) + a(1) + a(2) + a(3) + a(4)$.

In short, the extensions to our syntax is as follows:

$a(\ldots), b(\ldots), \ldots$	Single action with data parameters
$\sum_{d:D}$ _	Summation over variable d of type D
_ $\rightarrow$ _	Conditional operator
$P(\ldots), Q(\ldots), \ldots$	Process references with data parameters
$P(d : D, \ldots) =$ _	Process definition with parameters

These extensions must also be reflected in the semantics and axioms. Because of the fact that actions can now have data parameters, the semantics of operators ∇_V, ∂_H, τ_I and ρ_R must be reformulated such that data will not affect their behaviour (i.e. data is ignored). For the communication function we will only mention that, with data, we wish actions can only communicate if, and only if, they have equivalent parameters. Also, the rules for single actions and process references must be extended to include data parameters. As these changes are rather trivial, we will not give them here.

The additional rules and axioms are as given in Table 5 and Table 6. Note that we assume to have capture avoiding substitution and alpha conversion in $\doteq$.

Table 5
Additional *LoCo* Semantics for Data

$$\frac{t \xrightarrow{\alpha} \checkmark}{b \to t \xrightarrow{\alpha} \checkmark}\ \mathcal{A} \vDash b \qquad \frac{t \xrightarrow{\alpha} t'}{b \to t \xrightarrow{\alpha} t'}\ \mathcal{A} \vDash b$$

$$\frac{t[e/d] \xrightarrow{\alpha} \checkmark}{\sum_{d:D} t \xrightarrow{\alpha} \checkmark}\ e \in D \qquad \frac{t[e/d] \xrightarrow{\alpha} t'}{\sum_{d:D} t \xrightarrow{\alpha} t'}\ e \in D$$

Table 6
Extra *LoCo* Axioms for Data

C1	$\boldsymbol{t} \to x \doteq x$	*SUM1*	$\sum_{d:D} x \doteq x$
C2	$\boldsymbol{f} \to x \doteq \delta$	*SUM3*	$\sum_{d:D} p \doteq \sum_{d:D} p + p[e/d]$ *with* $e \in D$
V6	$\nabla_V(\sum_{d:D} p) \doteq \sum_{d:D} \nabla_V(p)$	*SUM4*	$\sum_{d:D}(p+q) \doteq \sum_{d:D} p + \sum_{d:D} q$
D6	$\partial_H(\sum_{d:D} p) \doteq \sum_{d:D} \partial_H(p)$	*SUM5*	$(\sum_{d:D} p) \cdot y \doteq \sum_{d:D}(p \cdot y)$
TI6	$\tau_I(\sum_{d:D} p) \doteq \sum_{d:D} \tau_I(p)$	*SUM6*	$(\sum_{d:D} p) \lfloor\!\lfloor y \doteq \sum_{d:D}(p \lfloor\!\lfloor y)$
R6	$\rho_R(\sum_{d:D} p) \doteq \sum_{d:D} \rho_R(p)$	*SUM7*	$(\sum_{d:D} p) \mid y \doteq \sum_{d:D}(p \mid y)$
G6	$\Gamma_C(\sum_{d:D} p) \doteq \sum_{d:D} \Gamma_C(p)$	*SUM7′*	$x \mid (\sum_{d:D} q) \doteq \sum_{d:D}(x \mid q)$

With $x, y \in V_P$ and $p, q \in T_{pc}$.

Of course, also with these extensions we want to have a sound and complete axiomatisation, as stated in the following theorems. Note that the completeness of the axiomatisation now depends on the completeness of derivability in the data algebra $\mathcal{A}$. As we do not explicitly consider this algebra, we say that our axiomatisation is relatively complete. This means that it is complete if we have completeness of derivability in $\mathcal{A}$.

Theorem 6.1 *The axiomatisation of LoCo with data is sound with respect to (rooted) branching bisimulation.*

Proof. This proof is very straightforward and therefore not given here. However, it can be found in [27].

Theorem 6.2 *The axiomatisation of LoCo with data (without $T1$ and $T2$) is relatively complete with respect to strong bisimulation.*

Proof. This proof is similar to completeness proof in [18]. Also, the full proof can be found in [27].

Theorem 6.3 *The axiomatisation of LoCo with data is relatively complete with respect to rooted branching bisimulation.*

Proof. The proof is similar to that in [12].

7 Examples

To illustrate the use of *LoCo*, we look at the following examples.

7.1 Petri net

The (coloured) Petri net in Figure 2 describes a little system that takes tokens (natural numbers) from place P_1 one at a time, performs a calculation on the token, and places it in P_4. We verify this behaviour by considering an interpretation of this Petri net in *LoCo*. We assume a true concurrency semantics for the Petri net. Although an interleaving semantics would make no difference here due to the structure of the example, the translation to *LoCo* itself does not put this restriction on the system.

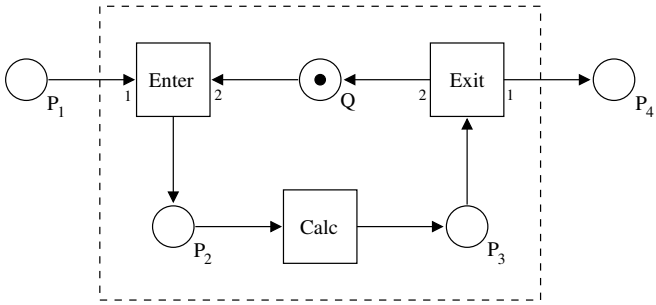


Fig. 2. A simple Petri net

Places are modelled by recursive processes parameterised with the contents of such a place. For places P_1, P_2, P_3 and P_4 this is a bag of natural numbers and for place Q a bag of $\mathbb{1}$ (with $\mathbb{1} = \{1\}$). Transitions are modelled by recursive processes without parameters (as transitions are memoryless) consisting of a single multiaction. Actions r_x and s_x are used to model the receiving respectively sending of a token by a place or transition X. These actions correspond to the incoming and outgoing arrows, respectively. If there are

two such incoming or outgoing arrows, the actions are labelled with the corresponding number from Figure 2. Thus transition *Enter* has an action r^1_{enter} with which he can receive a token from place P_1

The whole system is a parallel composition of the places and transitions enclosed by communication (modelling the links between places and transitions), hiding and restriction. Note that places P_1 and P_4 are enclosed by a blocking operator because they do not have an incoming or outgoing link, respectively. Also note that the behaviour of places is somewhat restricted here for simplicity; normally it would be possible to atomically take several tokens from one place, as well as putting new ones in it.

$$
\begin{aligned}
P_i(b : Bag(\mathbb{N})) &= \textstyle\sum_{n:\mathbb{N}} r_{p_i}(n) \cdot P_i(b \cup \{n\}) + \\
&\quad \textstyle\sum_{n:\mathbb{N}} n \in b \rightarrow s_{p_i}(n) \cdot P_i(b \setminus \{n\}) \\
Q(b : Bag(\mathbb{1})) &= \textstyle\sum_{i:\mathbb{1}} r_q(i) \cdot Q(b \cup \{i\}) + \\
&\quad \textstyle\sum_{i:\mathbb{1}} (i \in b) \rightarrow s_q(i) \cdot Q(b \setminus \{i\}) \\
Enter &= \textstyle\sum_{n:\mathbb{N}} r^1_{enter}(n) | r^2_{enter}(1) | s_{enter}(n) \cdot Enter \\
Calc &= \textstyle\sum_{n:\mathbb{N}} r_{calc}(n) | s_{calc}(f(n)) \cdot Calc \\
Exit &= \textstyle\sum_{n:\mathbb{N}} r_{exit}(n) | s^1_{exit}(n) | s^2_{exit}(1) \cdot Exit
\end{aligned}
$$

$$
\begin{aligned}
&PN(in : Bag(\mathbb{N}), out : Bag(\mathbb{N})) = \\
&\qquad \nabla_{\{\}}(\tau_{\{c\}}(\Gamma_{\{s_{p_1}|r^1_{enter}\rightarrow c,\ s_q|r^2_{enter}\rightarrow c,\ s_{enter}|r_{p_2}\rightarrow c,} \\
&\qquad\quad s_{p_2}|r_{calc}\rightarrow c,\ s_{calc}|r_{p_3}\rightarrow c,\ s_{p_3}|r_{exit}\rightarrow c,\ s^1_{exit}|r_{p_4}\rightarrow c,\ s^2_{exit}|r_q\rightarrow c\}(\\
&\qquad \partial_{\{r_{p_1}\}}(P_1(in)) \parallel P_2(\emptyset) \parallel P_3(\emptyset) \parallel \partial_{\{s_{p_4}\}}(P_4(out)) \parallel \\
&\qquad Q(\{1\}) \parallel Enter \parallel Calc \parallel Exit))))
\end{aligned}
$$

With basic expansion of the parallel operators and application of communication, hiding and restriction, we get the following result.

$$
\begin{aligned}
&PN(in : Bag(\mathbb{N}), out : Bag(\mathbb{N})) \doteq \\
&\qquad \textstyle\sum_{n:Nat} n \in in \rightarrow \tau \cdot PN(in \setminus \{n\}, out \cup \{f(n)\})
\end{aligned}
$$

As one can see, and could have expected, the behaviour of the system is the same as that of places P_1 and P_4 connected by transition *Calc*, which simply takes a number n from P_1 and puts $f(n)$ in P_4.

7.2 Components

In Figure 3, a system C, which checks whether components S_1 and S_2 return the same result for a given input, is depicted. Both components S_1 and S_2 take

an integer as input and return an integer as output. For computation they use components *Mul* and *Plus*, that multiply and add two integers, respectively. In addition, S_1 also uses *One*, that can always return a 1 on its output. Component *Cmp* takes two integers and returns $\boldsymbol{t}$ if they are equal (and $\boldsymbol{f}$ otherwise). Note that all computations occur instantaneous; a component produces output at the same time it takes it input.

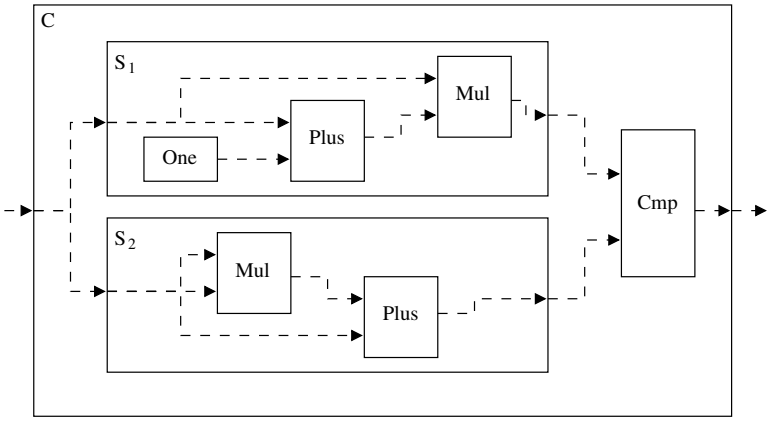


Fig. 3. A simple compositional system

All components are straightforwardly implemented as follows. Incoming arrows of a component X correspond to actions r_X and outgoing arrows correspond to actions s_X. If a component has more that one of such actions, then they are numbered from top to bottom as seen in Figure 3. Note that the individual components' specifications are completely self contained.

$$
\begin{array}{lcl}
\mathit{One} & = & s_{one}(1) \cdot \mathit{One} \\
\mathit{Mul} & = & \sum_{x:\mathbb{Z}} \sum_{y:\mathbb{Z}} r^1_{mul}(x) | r^2_{mul}(y) | s_{mul}(x * y) \cdot \mathit{Mul} \\
\mathit{Plus} & = & \sum_{x:\mathbb{Z}} \sum_{y:\mathbb{Z}} r^1_{plus}(x) | r^2_{plus}(y) | s_{plus}(x + y) \cdot \mathit{Plus} \\
S_1 & = & \nabla_{\{r_{s_1} | s_{s_1}\}}(\rho_{\{s_{mul} \to s_{s_1}\}}(\tau_{\{c\}}(\Gamma_{\{r^1_{mul} | r^1_{plus} \to r_{s_1},\ s_{one} | r^2_{plus} \to c,\ s_{plus} | r^2_{mul} \to c\}}(\\
 & & \quad \mathit{One} \parallel \mathit{Plus} \parallel \mathit{Mul})))) \\
S_2 & = & \nabla_{\{r_{s_2} | s_{s_2}\}}(\rho_{\{s_{plus} \to s_{s_2}\}} \tau_{\{c\}}(\Gamma_{\{r^1_{mul} | r^2_{mul} | r^2_{plus} \to r_{s_2},\ s_{mul} | r^1_{plus} \to c\}}(\mathit{Mul} \parallel \mathit{Plus})))) \\
\mathit{Cmp} & = & \sum_{x:\mathbb{Z}} \sum_{y:\mathbb{Z}} r^1_{cmp}(x) | r^2_{cmp}(y) | s_{cmp}(x = y) \cdot \mathit{Cmp} \\
C & = & \nabla_{\{r_c | s_c\}}(\rho_{\{s_{cmp} \to s_c\}}(\tau_{\{c\}}(\Gamma_{\{r_{s_1} | r_{s_2} \to r_c,\ s_{mul} | r^1_{cmp} \to c,\ s_{plus} | r^2_{cmp} \to c\}}(S_1 \parallel S_2 \parallel \mathit{Cmp}))))
\end{array}
$$

We can derive the following for S_1, S_2 and C.

$$
\begin{array}{lcl}
S_1 & \doteq & \sum_{x:\mathbb{Z}} r_{s_1}(x) | s_{s_1}(x * (x + 1)) \cdot S_1 \\
S_2 & \doteq & \sum_{x:\mathbb{Z}} r_{s_2}(x) | s_{s_2}(x * x + x) \cdot S_2 \\
C & \doteq & \sum_{x:\mathbb{Z}} r_c(x) | s_{cmp}(\boldsymbol{t}) \cdot C
\end{array}
$$

It is not possible to simple move the communication operators to the top level as would be required in language that have global communication instead

of local communication. If one would do so, it is impossible for, say, component *One* to know that he is communicating with the right *Plus* component. At least one instance of both the *Plus* and *Mul* components need to be changed to avoid conflicts in action names.

Although in this example the consequences of global communication are not extremely problematic, with bigger systems this becomes quite bothersome. Also, components S_1 and S_2 are typically developed independently. This means that one has to change the internals of S_1 and S_2 in order to use them safely in one system with global communication. It is clear that this is contrary to the ideas of component-based modelling.

8 Conclusion

We have introduced the process algebra *LoCo* that is truly compositional due to its local communication operator and the use of multiactions. It has a formal syntax, semantics and a sound and complete axiomatisation. We have included two small examples to illustrate the compositionality of *LoCo* and the ease with which Petri nets can be modelled in it.

As this work is mainly a basis for the mCRL2 language, future work will be continued in this context. This includes the addition of time, formal translations of Petri nets to mCRL2 and adapting existing proof techniques (such as those used with μCRL, for example) to the new setting.

References

[1] Austry, D. and G. Boudol, *Algèbre de processus et synchronisation*, Theoretical Computer Science **30** (1984), pp. 91–131.

[2] Baeten, J. and A. Basten, *Partial-order process algebra (and its relation to Petri nets)*, in: J. Bergstra, A. Ponse and S. Smolka, editors, *Handbook of Process Algebra*, Elsevier, 2001 pp. 769–872.

[3] Baeten, J. and J. Bergstra, *Non interleaving process algebra*, in: E. Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93)*, Lecture Notes in Computer Science **715** (1993), pp. 308–323.

[4] Baeten, J. and W. Weijland, "Process Algebra," Cambridge Tracts in Theoretical Computer Science **18**, Cambrdige University Press, 1990.

[5] Baeten, J. C. M. and C. Verhoef, *A congruence theorem for structured operational semantics with predicates*, in: E. Best, editor, *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93)*, Lecture Notes in Computer Science **715** (1993), pp. 477–492.

[6] Bernardo, M., P. Ciancarini and L. Donatiello, *Architecting families of software systems with process algebras*, ACM Trans. Softw. Eng. Methodol. **11** (2002), pp. 386–426.

[7] Best, E., R. Devillers and J. G. Hall, *The box calculus: a new causal algebra with multi-label communication*, in: G. Rozenberg, editor, *Advances in Petri Nets: The DEMON Project*, Lecture Notes in Computer Science **609** (1992), pp. 21–69.

[8] Best, E., R. Devillers and M. Koutny, "Petri net algebra," Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[9] Best, E., R. Devillers and M. Koutny, *A unified model for nets and process algebra*, in: J. Bergstra, A. Ponse and S. Smolka, editors, *Handbook of Process Algebra*, Elsevier, 2001 pp. 873–944.

[10] Bracciali, A., A. Brogi and F. Turini, *A framework for specifying and verifying the behaviour of open systems*, Journal of Logic an Algebraic Programming **63** (2005), pp. 215–240.

[11] Cuesta, C. E., P. de la Fuente, M. Barrio-Solórzano and M. E. Beato, *An "abstract process" approach to algebraic dynamic architecture description*, Journal of Logic an Algebraic Programming **63** (2005), pp. 177–214.

[12] Groote, J. and S. Luttik, *A complete axiomatisation of branching bisimulation for process algebras with alternative quantification over data*, Technical Report SEN-R9830, Centrum voor Wiskunde en Informatica (CWI) (1998).

[13] Groote, J., A. Mathijssen, M. van Weerdenburg and Y. Usenko, *From μCRL to mCRL2: Motivation and outline*, in: L. Aceto and A. D. Gordon, editors, *Proceedings of the Workshop Essays on Algebraic Process Calculi (APC 25)*, Electronic Notes in Theoretical Computer Science **162**, 2006, pp. 191–196.

[14] Groote, J. and A. Ponse, *The syntax and semantics of μCRL*, in: A. Ponse, C. Verhoef and S. van Vlijmen, editors, *Algebra of Communicating Processes, Workshops in Computing*, 1994, pp. 26–62.

[15] Groote, J. F., A. Mathijssen, M. Reniers, Y. Usenko and M. van Weerdenburg, *The formal specification language mcrl2*, in: E. Brinksma, D. Harel, A. Mader, P. Stevens and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings, 2007.

[16] ISO, *ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour*, Standard, International Standards Organization, Geneva, Switzerland (1987), first edition.

[17] Jensen, K., "Coloured Petri nets: basic concepts, analysis methods and practical use, vol. 2," Springer-Verlag, 1995.

[18] Luttik, B., "Choice Quantification in Process Algebra," Ph.D. thesis, University of Amsterdam (2002).

[19] Mauw, S. and M. Reniers, *A process algebra for interworkings*, in: J. Bergstra, A. Ponse and S. Smolka, editors, *Handbook of Process Algebra*, Elsevier, Amsterdam, 2001 pp. 1269–1327.

[20] Milner, R., "A Calculus of Communicating Systems," Springer-Verlag New York, Inc., 1982.

[21] Milner, R., *Calculi for synchrony and asynchrony*, Theoretical Computer Science **25** (1983), pp. 267–310.

[22] Milner, R., "Communicating and mobile systems: the π-calculus," Cambridge University Press, 1999.

[23] Norton, B., G. Lüttgen and M. Mendler, *A compositional semantic theory for synchronous component-based design.*, in: R. M. Amadio and D. Lugiez, editors, *CONCUR 2003 - Concurrency Theory, 14th International Conference*, Lecture Notes in Computer Science, 2003, pp. 453–467.

[24] Park, D., *Concurrency and automata on infinite sequences*, in: P. Deussen, editor, *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, Lecture Notes in Computer Science **104** (1981), pp. 167–183.

[25] van Glabbeek, R. J., *The linear time - branching time spectrum II*, in: E. Best, editor, *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, Lecture Notes in Computer Science **715** (1993), pp. 66–81.

[26] van Glabbeek, R. J. and W. P. Weijland, *Branching time and abstraction in bisimulation semantics*, J. ACM **43** (1996), pp. 555–600.

[27] van Weerdenburg, M., "GenSpect Process Algebra," Master's thesis, Eindhoven University of Technology (2004).

[28] van Weerdenburg, M., *Process algebra with local communication*, Technical Report 05/05, Eindhoven University of Technology (2005).