# Algebraic Process Verification

J. F. Groote
CWI, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands
Faculty of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
Email: JanFriso.Groote@cwi.nl

M. A. Reniers
Faculty of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
CWI, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands
Email: M.A.Reniers@tue.nl

**Abstract**

This chapter addresses the question how to verify distributed and communicating systems in an effective way from an explicit process algebraic standpoint. This means that all calculations are based on the axioms and principles of the process algebras. The first step towards such verifications is to extend process algebra (ACP) with equational data types which adds required expressive power to describe distributed systems. Subsequently, linear process operators, invariants, the cones and foci method, the composition of many similar parallel processes, and the use of confluence are explained, as means to verify increasingly complex systems. As illustration, verifications of the serial line interface protocol (SLIP) and the IEEE 1394 tree identify protocol are included.

*Keywords*: process algebra, abstract data type, verification, cones and foci proof method, linear process operator, confluence.

# 1 Introduction

The end of the seventies, beginning of the eighties showed the rise of process algebras such as CCS (Calculus of Communicating Systems) [Mil80], CSP (Communicating Sequential Processes) [Hoa85], and ACP (Algebra of Communicating Processes) [BK84, BW90]. The basic motivation for the introduction of process algebras was the need to describe and study programs that are dynamically interacting with their environment [Mil73, Bek71]. Before this time the mathematical view on programs was that of deterministic input/output transformers: a program starts with some input, runs for a while, and if it terminates, yields the output. Such programs can be characterised by partial functions from the input to the output. This view is quite suitable for simple 'batch processing', but it is clearly inadequate for commonly used programs such as operating systems, control systems or even text editors. These programs are constantly obtaining information from the environment that is subsequently processed and communicated. The development of distributed computing, due to the widespread availability of computer networks and computing equipment, makes that proper means to study interacting systems are needed.

Process algebras allow for a rather high level view on interacting systems. They assume that we do not know the true nature of such systems. They just regard all such systems as *processes*, objects in some mathematical domain. A process is best viewed as some object describing all the potential behaviour a program or system can execute. We only assume that certain (uninterpreted) actions $a$, $b$, $c, \ldots$ are processes, and that we can combine processes using a few operators, such as the sequential composition operator or the parallel composition operator. A number of axioms restrict these operators, just to guarantee that they satisfy the basic intuitions about them. This basically constitutes a process algebra: a domain of processes, and a set of operators satisfying certain axioms. Unfortunately, axioms are not always sufficient, and more general 'principles' are employed. All these principles, however, adhere to the abstract view on processes.

There are many approaches in the literature that, contrary to the process algebra approach, study processes as concrete objects, such as failure traces [Phi87], traces decorated with actions that cannot be executed at certain moments, Mazurkiewicz traces [Maz88], which contain an explicit indication of parallelism, event structures [Win87], Petri nets [Rei85, Jen92], objects in metric spaces [dBdV96], etc. etc. A partial overview of process models is given in [vG90, vG93]. A very useful perspective, which we employ for illustrations, is the view of a process as an automaton of which the transitions are labelled with actions. Each traversal through the automaton is a run of the process. This view allows one to compactly depict the operational behaviour of a process.

In this chapter, we want to increase our understanding of processes by manipulating them, proving their properties, or proving that certain processes have the same behaviour. We stress again that we do this strictly from the abstract process algebraic perspective. This means that all our calculations in this chapter are based on the axioms and principles.

There are two major difficulties one runs into if one tries to do process algebraic verifications in this way, applied to more than just trivial examples, namely restricted expressivity and lack of effective proof methodologies.

The basic reason for the expressivity problem is that basic process algebras cannot explicitly deal with data. Often this problem is circumvented by annotating data in the subscripts of process variables. A consequence of this is that the number of process variables becomes large or infinite, which is less elegant. Furthermore, it is impossible to communicate data taken from infinite data domains. This generally is dealt with by considering only finite, but sufficiently large data domains. A true problem, however, is that for larger verifications the majority of the calculations tend to shift to the data. The role of data as second class citizens hinders its effective manipulation. This has a direct repercussion on the size and difficulty of the systems that can be handled.

The other problem is that the axioms and principles are very elementary. This means that although there are very many ways to prove some property of processes, finding a particular proof turns out to be an immense task. What is called for are *proof methodologies*, i.e. recipes and guidelines that lead in reasonable time to relatively short proofs.

We have addressed the first problem by extending one of the basic process algebras, with data. The result is $\mu$CRL (micro Common Representation Language). Basically, it is a minimal extension to ACP with equational abstract data types. Special care has been taken to keep the language sufficiently small, to allow study of the language itself, and sufficiently rich to precisely and effectively describe all sorts of protocols, distributed algorithms and, in general all communicating systems.

In $\mu$CRL process variables and actions can be parameterised with data. Data can influence the course of a process through a conditional (if-then-else)construct, and alternative quantification is added to express choices over infinite data sorts. Recently, the language has been extended with features to express time [Gro97b], but time is not addressed in this chapter. $\mu$CRL has been the basis for the development of a proof theory [GP94], and a toolset [CWI00] allowing to simulate $\mu$CRL specifications and to perform all forms of finite state analysis on them. Using the toolset, it is even possible to do various forms of symbolic process manipulations, on the basis of the axioms, and currently this is an area under heavy investigation.

A lot of effort went into the specification and (manual) verification of various interactive systems [BG94a, GMS97, KS94, Lut97, KRR98, BBG97, GvdP96, FGK97]. When doing so, we developed a particular methodology of verification, culminating in the cones and foci technique [GS95], which enabled an increase in the order of magnitude of systems that could be analysed. As we strictly clung to the basic axioms and principles of process algebra, it was relatively easy (but still time consuming) to check our proofs using proof checkers such as Coq [DFH$^+$93], PVS [SOR93] and Isabelle [Pau90, Pau94] (for an overview see [GMvdP98]).

The first observation we ran into was that proving systems described in the full $\mu$CRL process syntax is inconvenient, despite the fact that the language is concise. Therefore, a normal form that is both sufficiently powerful to represent all systems denotable in $\mu$CRL and that is very straightforward was required. We took Linear Process Operators or Linear Process Equations as the normal form. This format resembles I/O automata [Lyn96], extended finite state machines [IT94] or Unity processes [CM89]. It is explained in Section 3.1.

An obvious verification problem that we often encounter is to prove an implementation adhering to its specification. We found this to be hard for basically the same reasons in all instances we studied. By equivalence we generally understand rooted branching bisimilarity. In this case the verification task is roughly that visible actions in the implementation should be matched by visible actions in corresponding states in the specification and vice versa. The difficulty is that often an action in the specification can only be matched in the implementation by first doing a large number of internal steps. The implementation contains many cone-like structures as sketched in Figure 1, where internal actions in the cone precede the external actions at the edge of the cone. The cones and foci method employs these structures, as summarised in the generalised equality theorem (see Section 3.3), reducing the proof of equality to a proof of properties of data parameters that are relatively easy to handle.

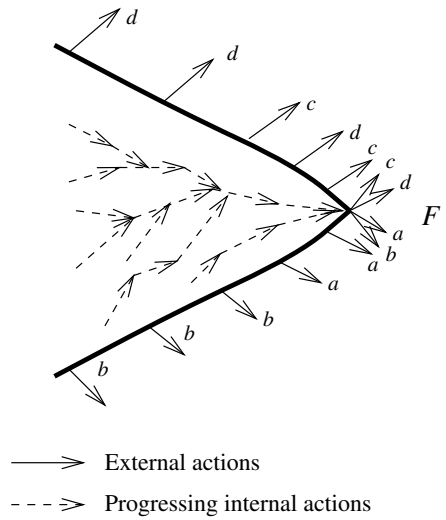



Figure 1: A cone and a focus point.

In this method the notion of invariant has been introduced. Despite the fact that invariants are the most important technical means to carry out sequential program verification, they were virtually absent in process algebra. Although it can be shown that formally invariants are not needed, in the sense that each process algebraic proof using invariants can be transformed into one without explicit use of this notion, we believe that invariants are

important. The reason for this is that it allows to split a proof in on the one hand finding appropriate invariants, and on the other hand proving an equivalence or property.

Another difficulty is that one often needs to prove properties of distributed systems that consist of an arbitrary number, say $n$, similar processes. It turns out that calculating the parallel composition of these $n$ processes with induction on $n$ is cumbersome (see e.g. [KS94]). The reason for this is that it requires to describe the behaviour of only a subset of the $n$ processes. However, if one looks at the problem from a different angle, the parallel composition becomes a simple mechanical procedure. This is exactly the topic of Section 5.

Section 7 deals with confluence which is one of the most obvious structures occurring in distributed systems. In Milner's seminal book on process algebra [Mil80], a full chapter was devoted to the subject. In Section 7 it is shown how confluence can be used to simplify the behaviour of a system considerably, after which it is much easier to understand and analyse it.

We feel slightly unsatisfied that only the topics mentioned above are addressed in this chapter. There is much more known and to be known about process verification. We could not include the large number of potentially effective techniques about which ideas exist, but which have not yet developed sufficiently in the process algebra context to be included. One may think about classifying distributed systems in several categories with similar structures, the use of symmetry in distributed systems, and the use of history and especially prophecy variables. Even if we would have attempted to include such ideas, we would soon be incomplete, as we feel that algebraic process verification is only at the brink of its development.

The next section starts with a thorough explanation of the language $\mu$CRL. We subsequently address the topics mentioned above. Interleaved with these we prove two distributed systems correct, as an illustration of the method.

The material presented in this chapter is based on a number of publications. Section 2 is based on [GP95]. The cones and foci method and the general equality theorem presented in Section 3 are taken from [GS95]. The verification of the SLIP protocol in Section 4 is slightly adapted from [GMvdP98]. The linearisation of a number of similar processes presented in Section 5 is taken from [Gro97a]. The example of the Tree Identify Protocol of IEEE 1394 (Section 6) is taken from [SvdZ98]. Section 7 on confluence is based on [GS96].

**Acknowledgements.** We would like to thank Harm van Beek, Michiel van Osch, Piet Rodenburg, and Mark van der Zwaag for their valuable comments on several versions of this chapter.

# 2 Process algebra with data: $\mu$CRL

In this section we describe $\mu$CRL, which is a process algebra with data. The process algebra $\mu$CRL is used in different contexts and for different purposes. On the one hand it is used as a formal specification language with a strict syntax and (static) semantics. As such it can be used as input for a formal analysis toolset. On the other hand it is a mathematical notation, with the flexibility to omit obvious definitions, to only sketch less relevant parts, introduce convenient ad hoc notation, etc. In this section we stick quite closely to $\mu$CRL as a formal language. In the subsequent sections we are much less strict, and take a more mathematical approach.

## 2.1 Describing data types in $\mu$CRL

In $\mu$CRL there is a simple, yet powerful mechanism for specifying data. We use equationally specified abstract data types with an explicit distinction between constructor functions and 'normal' functions. The advantage of having such a simple language is that it can easily be explained and formally defined. Moreover, all properties of a data type must be defined explicitly, and henceforth it is clear which assumptions can be used when proving properties of data or processes. A disadvantage is of course that even the simplest data types must be specified each time, and that there are no high level constructs that allow compact specification of complex data types. Still, thus far these shortcomings have not outweighed the advantage of the simplicity of the language.

Each data type is declared using the keyword **sort**. Therefore, a data type is also called a data sort. Each declared sort represents a non-empty set of data elements. Declaring the sort of the booleans is simply done by:

**sort** **Bool**

Because booleans are used in the if-then-else construct in the process language, the sort **Bool** must be declared in every $\mu$CRL specification.

Elements of a data type are declared by using the keywords **func** and **map**. Using **func** one can declare all elements in a data type defining the structure of the data type. E.g. by

**sort** **Bool**
**func** $\mathsf{t}, \mathsf{f} :\rightarrow$ **Bool**

one declares that `t` (true) and `f` (false) are the only elements of sort **Bool**. We say that `t` and `f` are the constructors of sort **Bool**. Not only the sort **Bool**, but also its elements `t`

and $\mathsf{f}$ must be declared in every specification; they must be distinct, and the only elements of **Bool**. This is expressed in axioms Bool1 and Bool2 in Table 1. In axiom Bool2 and elsewhere we use a variable $b$ that can only be instantiated with data terms of sort **Bool**. If in a specification $\mathsf{t}$ and $\mathsf{f}$ can be proven equal, for instance if the specification contains an equation $\mathsf{t} = \mathsf{f}$, we say that the specification is inconsistent and it looses any meaning. We often write $\phi$ and $\neg\phi$ instead of $\phi = \mathsf{t}$ and $\phi = \mathsf{f}$, respectively.

| | |
|---|---|
| Bool1 | $\neg(\mathsf{t} = \mathsf{f})$ |
| Bool2 | $\neg(b = \mathsf{t}) \rightarrow b = \mathsf{f}$ |

Table 1: Basic axioms for **Bool**.

It is now easy to declare the natural numbers using the constructors 0 and successor $S$.

**sort** $Nat$
**func** $0 :\rightarrow Nat$
$S : Nat \rightarrow Nat$

This says that each natural number can be written as 0 or the result of a number of applications of the successor function to 0.

If a sort $D$ is declared without any constructor with target sort $D$, then it is assumed that $D$ may be arbitrarily large. In particular $D$ may contain elements that cannot be denoted by terms. This can be extremely useful, for instance when defining a data transfer protocol, that can transfer data elements from an arbitrary domain $D$. In such a case it suffices to declare in $\mu$CRL:

**sort** $D$

The keyword **map** is used to declare additional functions for a domain of which the structure is already given. For instance declaring a function $\wedge$ on the booleans, or, declaring the $+$ on natural numbers, can be done by adding the following lines to a specification in which $Nat$ and **Bool** have already been declared:

**map** $\wedge : \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$
$+ : Nat \times Nat \rightarrow Nat$

By adding plain equations between terms assumptions about the functions can be added. For the two functions declared above, we could add the equations:

**var** $x$:**Bool**
$n, n'$:*Nat*
**rew** $x \wedge \mathsf{t} = x$
$x \wedge \mathsf{f} = \mathsf{f}$
$n + 0 = n$
$n + S(n') = S(n + n')$

Note that before each group of equations starting with the keyword **rew** we must declare the variables that are used.

The machine readable syntax of $\mu$CRL only allows prefix notation for functions, but we use infix or even postfix notation, if we believe that this increases readability. Moreover, we use common mathematical symbols such as $\wedge$ and $+$ in data terms, which are also not allowed by the syntax of $\mu$CRL, for the same reason.

Functions may be overloaded, as long as every term has a unique sort. This means that the name of the function together with the sort of its arguments must be unique. E.g. it is possible to declare $max : Nat \times Nat \rightarrow Nat$ and $max : \mathbf{Bool} \times \mathbf{Bool} \rightarrow \mathbf{Bool}$, but is not allowed to declare a function $f : \mathbf{Bool} \rightarrow \mathbf{Bool}$ and $f : \mathbf{Bool} \rightarrow Nat$. Actually, the overloading rule holds in general in $\mu$CRL. The restrictions on declarations are such that every term is either an action, a process name or a data term, and if it is a data term, it has a unique sort.

Although we have that every term of a data sort equals a term that is only built from the constructor functions this does not mean that we always know which constructor term this will be. For example, if we introduce an additional function 2 for the sort *Nat* by means of the declaration

**map** $2 :\rightarrow Nat$

this does not give us which constructor term equals the constant 2. This information can be added explicitly by adding an equation

**rew** $2 = S(S(0))$

When we declare a sort $D$, it must be nonempty. Therefore, the following declaration is invalid.

**sort** $D$
**func** $f : D \rightarrow D$

It declares that $D$ is a domain in which all the terms have the form $f(f(f(\ldots)))$, i.e. an infinite number of applications of $f$. Such terms do not exist, and therefore $D$ must be empty. This problem can also occur with more than one sort. E.g. sorts $D$ and $E$ with constructors from $D$ to $E$ and $E$ to $D$. Fortunately, it is easy to detect such problems and therefore it is a static semantic constraint that such empty sorts are not allowed (see [GP95]).

In proving the equality of data terms we can use the axioms, induction on the constructor functions of the data types and all deduction rules of equational logic. An abstract data type can be used to prove elementary properties. We explain here how we can prove data terms equal with induction, and we also show how we can prove data terms to be nonequal.

An easy and very convenient axiom is Bool2. It says that if a boolean term $b$ is not equal to $\mathsf{t}$, it must be equal to $\mathsf{f}$ or in other words that there are at most two boolean values. Applying this axiom boils down to a case distinction, proving a statement for the values $\mathsf{t}$ and $\mathsf{f}$, and concluding that the property must then universally hold. We refer to this style of proof by the phrase 'induction on booleans'.

A typical example is the proof of $b \wedge b = b$. Using induction on **Bool**, it suffices to prove that this equality holds for $b = \mathsf{t}$ and $b = \mathsf{f}$. In other words, we must show that $\mathsf{t} \wedge \mathsf{t} = \mathsf{t}$ and $\mathsf{f} \wedge \mathsf{f} = \mathsf{f}$. These are trivial instances of the defining axioms for $\wedge$ listed above.

Note that the sort **Bool** is the only sort for which we explicitly state that the constructors $\mathsf{t}$ and $\mathsf{f}$ are different. For other sorts, like *Nat*, there are no such axioms.

The division between constructors and mappings gives us general induction principles. If a sort $D$ is declared with a number of constructors, then we may assume that every term of sort $D$ can be written as the application of a constructor to a number of arguments chosen from the respective argument sorts. So, if we want to prove a property $p(d)$ for all $d$ of sort $D$, we only need to provide proofs for $p(c_n(d_1, \ldots, d_n))$ for each $n$-ary constructor $c_n{:}S_1 \times \cdots \times S_n \to D$ and each $d_i$ a term of sort $S_i$. If any of the arguments of $c_n$, say argument $d_j$, is of sort $D$ then, as $d_j$ is smaller than $d$, we may use that $p(d_j)$. If we apply this line of argumentation, we say we apply induction on $D$.

Suppose we have declared the natural numbers with constructors zero and successor, as done above. We can for instance derive that $0 + n = n$ for all $n$. We apply induction on *Nat*. First, we must show that $0 + 0 = 0$, considering the case where $n = 0$. This is a trivial instance of the first axiom on addition. Secondly, we must show $0 + S(n') = S(n')$, assuming that $n$ has the form $S(n')$. In this case we may assume that the property to be proven holds already for $n'$, i.e. $0 + n' = n'$. Then we obtain:

$$0 + S(n') = S(0 + n') = S(n').$$

As an example, we define a sort *Queue* on an arbitrary non-empty domain $D$, with an empty queue $[\,]$, and *in* to insert an element of $D$ into the queue. The arbitrary non-empty

domain is obtained by the specification of sort $D$ without constructors.

**sort** $D$, $Queue$
**func** $[\,] :\rightarrow Queue$
$in : D \times Queue \rightarrow Queue$

We extend this with auxiliary definitions *toe* to get the first element from a queue, *untoe* to remove the first element from a queue, *isempty* to check whether a queue is empty and $++$ to concatenate two queues.

**map** $toe : Queue \rightarrow D$
$untoe : Queue \rightarrow Queue$
$isempty : Queue \rightarrow \mathbf{Bool}$
**var** $d, d'{:}D$
$q, q'{:}Queue$
**rew** $toe(in(d, [\,])) = d$
$toe(in(d, in(d', q))) = toe(in(d', q))$
$untoe(in(d, [\,])) = [\,]$
$untoe(in(d, in(d', q))) = in(d, untoe(in(d', q)))$
$isempty([\,]) = \mathsf{t}$
$isempty(in(d, q)) = \mathsf{f}$
$[\,]{+}{+}q = q$
$in(d, q){+}{+}q' = in(d, q{+}{+}q')$

A queue $q_1$ from which the last element has been removed can be given by $untoe(q_1)$ and a queue $q_2$ into which the last element of $q_1$ has been inserted is given by $in(toe(q_1), q_2)$. Now we prove

$$\neg isempty(q_1) \rightarrow untoe(q_1){+}{+}in(toe(q_1), q_2) = q_1{+}{+}q_2.$$

Suppose that $\neg isempty(q_1)$. We prove the proposition by induction on the structure of queue $q_1$.

*Base.* Suppose that $q_1 = [\,]$. Then $isempty([\,]) = \mathsf{t}$, which contradicts the assumption that $\neg isempty(q_1)$.

*Induction step.* Suppose that $q_1 = in(d, q_1')$ for some $d{:}D$ and $q_1'{:}Queue$. By induction we have $\neg isempty(q_1') \rightarrow untoe(q_1'){+}{+}in(toe(q_1'), q_2) = q_1'{+}{+}q_2$. Then we can distinguish the following two cases for $q_1'$:

- $q_1' = [\,]$. In this case we have

$$\begin{aligned}
&untoe(q_1){+\!+}in(toe(q_1), q_2)\\
=&untoe(in(d, [\,]))+\!+in(toe(in(d, [\,])), q_2)\\
=&[\,]+\!+in(d, q_2)\\
=&in(d, q_2)\\
=&in(d, [\,]+\!+q_2)\\
=&in(d, [\,])+\!+q_2\\
=&q_1+\!+q_2.
\end{aligned}$$

- $q_1' = in(d', q_1'')$. In this case we have

$$\begin{aligned}
&untoe(q_1)+\!+in(toe(q_1), q_2)\\
=&untoe(in(d, in(d', q_1'')))+\!+in(toe(in(d, in(d', q_1''))), q_2)\\
=&in(d, untoe(in(d', q_1'')))+\!+in(toe(in(d', q_1'')), q_2)\\
=&in(d, untoe(q_1')+\!+in(toe(q_1'), q_2))\\
=&in(d, q_1'+\!+q_2)\\
=&in(d, q_1')+\!+q_2\\
=&q_1+\!+q_2.
\end{aligned}$$

  Note that we used that $untoe(q_1')+\!+in(toe(q_1'), q_2) = q_1'+\!+q_2$. This is allowed as we can derive that $isempty(q_1') = isempty(in(d', q_1'')) = \mathsf{f}$.

Using the previous proposition we can easily prove that

$$\neg isempty(q) \rightarrow untoe(q)+\!+in(toe(q), [\,]) = q$$

for all $q{:}Queue$. For if we take $q_1 = q$ and $q_2 = [\,]$ we obtain:

$$\neg isempty(q) \rightarrow untoe(q)+\!+in(toe(q), [\,]) = q+\!+[\,].$$

Assuming that we can prove $q+\!+[\,] = q$, it is not hard to see that we thus have obtained $\neg isempty(q) \rightarrow untoe(q)+\!+in(toe(q), [\,]) = q$. The property $q+\!+[\,] = q$ for all $q{:}Queue$ can be proven with induction on the structure of $q$:

*Base.* Suppose that $q = [\,]$. Clearly $q+\!+[\,] = [\,]+\!+[\,] = [\,] = q$.

*Induction step.* Suppose that $q = in(d, q')$. By induction we have $q'+\!+[\,] = q'$. Then $q+\!+[\,] = in(d, q')+\!+[\,] = in(d, q'+\!+[\,]) = in(d, q') = q$.

In $\mu$CRL it is possible to establish when two data terms are not equal. This is for instance required in order to establish that two processes cannot communicate. There is a characteristic way of proving that terms are not equal, namely by assuming that they are equal, and showing that this implies $\mathsf{t} = \mathsf{f}$, contradicting axiom Bool1.

We give an example showing that the natural numbers zero (0) and one ($S(0)$) are not equal. We assume that the natural numbers with a 0 and successor function $S$ are appropriately declared. In order to show zero and one different, we need a function that relates *Nat* to **Bool**. Note that if there is no such function, there are models of the data type *Nat* where zero and one are equal. For our function we choose 'less than or equal to', notation $\leq$, on the natural numbers, defined as follows:

**map** $\leq: Nat \times Nat \to \mathbf{Bool}$
**var** $n, m{:}Nat$
**rew** $0 \leq n = \mathsf{t}$
$S(n) \leq 0 = \mathsf{f}$
$S(n) \leq S(m) = n \leq m$

Now assume $0 = S(0)$. Clearly, $0 \leq 0 = \mathsf{t}$. On the other hand, using the assumption, we also find $0 \leq 0 = S(0) \leq 0 = \mathsf{f}$. So, we can prove $\mathsf{t} = \mathsf{f}$. Hence, we may conclude $0 \neq S(0)$.

This finishes the most important aspects of the data types. There are several standard libraries available [vW95, MV90] of which some also contain numerous provable identities. The general theory about abstract data types is huge, see for instance [EM85].

## 2.2 Describing processes in $\mu$CRL

### 2.2.1 Actions

Actions are abstract representations of events in the real world that is being described. For instance sending the number 3 can be described by $send(3)$ and boiling food can be described by $boil(food)$ where 3 and *food* are terms declared by a data type specification. An action consists of an action name possibly followed by one or more data terms within brackets. Actions are declared using the keyword **act** followed by an action name and the sorts of the data with which it is parameterised. Below, we declare the action name *timeout* without parameters, an action $a$ that is parameterised with booleans, and an action $b$ that is parameterised with pairs of natural numbers and data elements. The set of all action names that are declared in a $\mu$CRL specification is denoted by $\mathsf{Act}$.

**act** *timeout*
$a$ : **Bool**
$b$ : $Nat \times D$

In accordance with mainstream process algebras, actions in $\mu$CRL are considered to be atomic. If an event has a certain positive duration, such as boiling food, then it is most

appropriate to consider the action as the beginning of the event. If the duration of the event is important, separate actions for the beginning and termination of the event can be used.

In the tables with axioms we use the letters $a$ and $a'$ for action names, and in order to be concise, we give each action a single argument, although in $\mu$CRL these actions may have zero or more than one argument. The letter $c$ is used for actions with an argument, and for the constants $\delta$ and $\tau$, which are explained in Section 2.2.3 and Section 2.2.8 respectively.

### 2.2.2 Alternative and sequential composition

The two elementary operators for the construction of processes are the *sequential composition* operator, written as $p \cdot q$ and the *alternative composition* operator, written as $p + q$. The process $p \cdot q$ first performs the actions of $p$, until $p$ terminates, and then continues with the actions in $q$. It is common to omit the sequential composition operator in process expressions. The process $p + q$ behaves like $p$ or $q$, depending on which of the two performs the first action. Using the actions declared above, we can describe that $a(3, d)$ must be performed, except if a time out occurs, in which case $a(\mathtt{t})$ must happen.

$$a(3, d) + \mathit{timeout} \cdot a(\mathtt{t})$$

Observe that the sequential composition operator binds stronger than the alternative composition operator.

| | |
|---|---|
| A1 | $x + y = y + x$ |
| A2 | $x + (y + z) = (x + y) + z$ |
| A3 | $x + x = x$ |
| A4 | $(x + y) \cdot z = x \cdot z + y \cdot z$ |
| A5 | $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ |

Table 2: Basic axioms for $\mu$CRL.

In Table 2 axioms A1-A5 are listed describing the elementary properties of the sequential and alternative composition operators. For instance, the axioms A1, A2 and A3 express that + is commutative, associative and idempotent. In these and other axioms we use variables $x$, $y$ and $z$ that can be instantiated by process terms.

For processes we use the shorthand $x \subseteq y$ for $x + y = y$ and we write $x \supseteq y$ for $y \subseteq x$. This notation is called *summand inclusion*. It is possible to divide the proof of an equality into proving two inclusions, as the following lemma shows.

**Lemma 2.1** *For arbitrary $\mu$CRL-terms $x$ and $y$ we have: if $x \subseteq y$ and $y \subseteq x$, then $x = y$.*

**Proof.** Suppose $x \subseteq y$ and $y \subseteq x$. By definition we thus have (1) $x + y = y$ and (2) $y + x = x$. Thus we obtain: $x \overset{(2)}{=} y + x \overset{\text{A1}}{=} x + y \overset{(1)}{=} y$. ⊠

### 2.2.3 Deadlock

The language $\mu$CRL contains a constant $\delta$, expressing that no action can be performed, for instance in case a number of computers are waiting for each other, and henceforth not performing anything. This constant is called *deadlock*. A typical property for $\delta$ is $p+\delta = p$; the choice in $p+q$ is determined by the first action performed by either $p$ or $q$, and therefore one can never choose for $\delta$. In other words, as long as there are alternatives deadlock is avoided. In Table 3 the axioms A6 and A7 characterise the main properties of $\delta$.

| | |
|---|---|
| A6 | $x + \delta = x$ |
| A7 | $\delta \cdot x = \delta$ |

Table 3: Axioms for deadlock.

### 2.2.4 Process declarations

Process expressions are expressions built upon actions indicating the order in which the actions may happen. In other words, a process expression represents the potential behaviour of a certain system.

In a $\mu$CRL specification process expressions appear at two places. First, there can be a single occurrence of an initial declaration, of the form

**init** $p$

where $p$ is a process expression indicating the initial behaviour of the system that is being described. The **init** section may be omitted, in which case the initial behaviour of the system is left unspecified.

The other place where process expressions may occur are in the right hand side of process declarations, which have the form:

**proc** $X(x_1{:}s_1, \ldots, x_n{:}s_n) = p$

Here $X$ is the process name, the $x_i$ are variables, not clashing with the name of a constant function or a parameterless process or action name, and the $s_i$ are sort names. In this rule, process $X(x_1, \ldots, x_n)$ is declared to have the same behaviour as the process expression $p$.

The equation in a process declaration must be considered as an equation in the ordinary mathematical sense. This means that in a declaration such as the one above an occurrence of $X(u_1, \ldots, u_n)$ may be replaced by $p(u_1/x_1, \ldots, u_n/x_n)$, or vice versa, $p(u_1/x_1, \ldots, u_n/x_n)$ may be replaced by $X(u_1, \ldots, u_n)$.

An example of a process declaration is the following clock process which repeatedly performs the action *tick* and displays the current time. In this example and also in later examples we assume the existence of a sort *Nat* with additional operators which represents the natural numbers. We simply write 1 instead of $S(0)$, 2 instead of $S(S(0))$, etc. Furthermore, we assume that the standard functions on naturals are defined properly. Examples of such functions are $+$, $\leq$, $<$, $>$, etc.

**act** $tick$
$display : Nat$
**proc** $Clock(t{:}Nat) = tick \cdot Clock(t+1) + display(t) \cdot Clock(t)$
**init** $Clock(0)$

### 2.2.5 Conditionals

The process expression $p \lhd b \rhd q$ where $p$ and $q$ are process expressions, and $b$ is a data term of sort **Bool**, behaves like $p$ if $b$ is equal to $\mathsf{t}$ (true) and behaves like $q$ if $b$ is equal to $\mathsf{f}$ (false). This operator is called the *conditional operator*, and operates precisely as an *then_if_else* construct. Through the conditional operator data influences process behaviour. For instance a counter, that counts the number of $a$ actions that occur, issuing a $b$ action and resetting the internal counter after 10 $a$'s, can be described by:

**proc** $Counter(n{:}Nat) = a \cdot Counter(n+1) \lhd n < 10 \rhd b \cdot Counter(0)$

The conditional operator is characterised by the axioms C1 and C2 in Table 4. All the properties of conditionals that we need are provable from these axioms and Bool1, Bool2.

The conditional operator binds stronger than the alternative composition operator and weaker than the sequential composition operator.

| | |
|---|---|
| C1 | $x \lhd \mathsf{t} \rhd y = x$ |
| C2 | $x \lhd \mathsf{f} \rhd y = y$ |

Table 4: Axioms for conditionals.

**Lemma 2.2** *The following identities hold for arbitrary $\mu$CRL-terms $x$, $y$, $z$ and for arbitrary boolean terms $b$, $b_1$, $b_2$.*

1. $x \lhd b \rhd y = x \lhd b \rhd \delta + y \lhd \neg b \rhd \delta$;
2. $x \lhd b_1 \vee b_2 \rhd \delta = x \lhd b_1 \rhd \delta + x \lhd b_2 \rhd \delta$;
3. $(b = \mathsf{t} \rightarrow x = y) \rightarrow x \lhd b \rhd z = y \lhd b \rhd z$.

### 2.2.6 Alternative quantification

The *sum operator* or *alternative quantification* $\sum_{d:D} P(d)$ behaves like $P(d_1) + P(d_2) + \cdots$, i.e. as the possibly infinite choice between $P(d_i)$ for any data term $d_i$ taken from $D$. This is generally used to describe a process that is reading some input. E.g. in the following example we describe a single-place buffer, repeatedly reading a natural number $n$ using action name $r$, and then delivering that value via action name $s$.

**proc** $\mathit{Buffer} = \sum_{n:Nat} r(n) \cdot s(n) \cdot \mathit{Buffer}$

Note that alternative quantification binds stronger than the alternative composition operator and weaker than the conditional operator.

In Table 5 the axioms for the sum operator are listed. The sum operator $\sum_{d:D} p$ is a difficult operator, because it acts as a binder just like the lambda in the lambda calculus (see e.g. [Bar81]). This introduces a range of intricacies with substitutions. In order to avoid having to deal with these explicitly, we allow the use of explicit lambda operators and variables representing functions from data to process expressions.

| | |
|---|---|
| SUM1 | $\sum_{d:D} x = x$ |
| SUM3 | $\sum X = \sum X + Xd$ |
| SUM4 | $\sum_{d:D}(Xd + Yd) = \sum X + \sum Y$ |
| SUM5 | $(\sum X) \cdot x = \sum_{d:D}(Xd \cdot x)$ |
| SUM11 | $(\forall_{d:D}\ (Xd = Yd)) \rightarrow \sum X = \sum Y$ |

Table 5: Axioms for alternative quantification.

In the tables the variables $x$, $y$ and $z$ may be instantiated with process expressions and the capital variables $X$ and $Y$ can be instantiated with functions from some data sort to process expressions. The sum operator $\sum$ expects a function from a data sort to a process expression, whereas $\sum_{d:D}$ expects a process expression. Moreover, we take $\sum_{d:D} p$ and $\sum \lambda d{:}D.p$ to be equivalent.

When we substitute a process expression $p$ for a variable $x$ or a function $\lambda d{:}D.p$ for a variable $X$ in the scope of a number of sum operators, no variable in $p$ may become bound by any of these sum operators. So, we may not substitute the action $a(d)$ for $x$ in the left hand side of SUM1 in Table 4, because this would cause $d$ to become bound by the sum operator. So, SUM1 is a concise way of saying that if $d$ does not appear in $p$, then we may omit the sum operator in $\sum_{d:D} p$.

As another example, consider axiom SUM4. It says that we may distribute the sum operator over a plus, even if the sum binds a variable. This can be seen by substituting for $X$ and $Y$ $\lambda d{:}D.a(d)$ and $\lambda d{:}D.b(d)$, where no variable becomes bound. After $\beta$-reduction, the left hand side of SUM4 becomes $\sum_{d:D}(a(d)+b(d))$ and the right hand side is $\sum_{d:D} a(d)+\sum_{d:D} b(d)$. In conformity with the $\lambda$-calculus, we allow $\alpha$-conversion in the sum operator, and do not state this explicitly. Hence, we consider the expressions $\sum_{d:D} p(d)$ and $\sum_{e:D} p(e)$ as equal.

The axiom SUM3 allows to split single summand instances from a given sum. For instance the process expressions $\sum_{n:Nat} a(n)$ and $\sum_{n:Nat} a(n) + a(2)$ are obviously the same, as they allow an $a(n)$ action for every natural number $n$. Using SUM3 we can prove them equal. Instantiate $X$ with $\lambda n.a(n)$ and $d$ with 2. We obtain:

$$\sum \lambda n.a(n) = \sum \lambda n.a(n) + (\lambda n.a(n))2.$$

By $\beta$-reduction this reduces to $\sum_{n:Nat} a(n) = \sum_{n:Nat} a(n) + a(2)$.

We show how we can eliminate a finite sum operator in favour of a finite number of alternative composition operators. Such results always depend on the fact that a data

type is defined using constructors. Therefore, we need induction in the proof, which makes it appear quite involved. This apparent complexity is increased by the use of axioms SUM3 and SUM11. Consider the equality

$$\sum_{n:Nat} r(n) \triangleleft n \leq 2 \triangleright \delta = r(0) + r(1) + r(2), \tag{1}$$

assuming that the natural numbers together with the $\leq$ relation have been appropriately defined. The result follows in a straightforward way by the following lemma that we prove first.

**Lemma 2.3** *For all m:Nat we find (S is the successor function):*

$$\sum_{n:Nat} Xn = X0 + \sum_{m:Nat} XS(m).$$

**Proof.** Using Lemma 2.1 we can split the proof into two summand inclusions.

$\subseteq$) We first prove the following statement with induction on $n$:

$$Xn \subseteq X0 + \sum_{m:Nat} XS(m).$$

- $n = 0$) Trivial using A3.
- $n = S(n')$)

$$\begin{array}{l} X0 + \sum_{m:Nat} XS(m) \overset{\text{SUM3}}{=} \\ X0 + \sum_{m:Nat} XS(m) + XS(n') \supseteq \\ Xn. \end{array}$$

So the statement has been proven without assumptions on $n$ (i.e. for all $n$). Hence, application of SUM11, SUM4 and SUM1 yields:

$$\sum_{n:Nat} Xn \subseteq X0 + \sum_{m:Nat} XS(m),$$

as was to be shown.

$\supseteq$) Using SUM3 it immediately follows that for all $m$

$$\sum_{n:Nat} Xn \supseteq X0 + XS(m).$$

So, SUM11, SUM4 and SUM1 yield:

$$\sum_{n:Nat} Xn \supseteq X0 + \sum_{m:Nat} XS(m).$$

$\boxtimes$

Equation (1) can now easily be proven by:

$$\begin{array}{l}\sum_{n:Nat} r(n) \lhd n \leq 2 \rhd \delta \overset{\text{Lemma 2.3}}{=} \\ r(0) \lhd 0 \leq 2 \rhd \delta + \sum_{n':Nat} r(n'+1) \lhd n'+1 \leq 2 \rhd \delta \overset{\text{Lemma 2.3}}{=} \\ r(0) + r(1) \lhd 1 \leq 2 \rhd \delta + \sum_{n'':Nat} r(n''+2) \lhd n''+2 \leq 2 \rhd \delta \overset{\text{Lemma 2.3}}{=} \\ r(0) + r(1) + r(2) \lhd 2 \leq 2 \rhd \delta + \sum_{n''':Nat} r(3+n''') \lhd n'''+3 \leq 2 \rhd \delta = \\ r(0) + r(1) + r(2)\end{array}$$

All the identities on data that we have used in the proof above can be proved from the axioms on natural numbers in Section 2.1.

An important law is *sum elimination*. It states that the sum over a data type from which only one element can be selected can be removed. This lemma occurred for the first time in [GK94]. Note that we assume that we have a function *eq* available, reflecting equality between terms.

**Lemma 2.4 (Sum elimination)** *Let $D$ be a sort and $eq : D \times D \to$ **Bool** a function such that for all $d, e{:}D$ it holds that $eq(d, e) = \mathtt{t}$ iff $d = e$. Then*

$$\sum_{d:D} Xd \lhd eq(d, e) \rhd \delta = Xe.$$

**Proof.** According to Lemma 2.1 it suffices to prove summand inclusion in both directions.

$\subseteq$) Using Lemma 2.2.2 above we find:

$$Xe = Xe \lhd eq(d, e) \rhd \delta + Xe \lhd \neg eq(d, e) \rhd \delta.$$

Using SUM11 and SUM4 we find:

$$\sum_{d:D} Xe = \sum_{d:D} Xd \lhd eq(d, e) \rhd \delta + \sum_{d:D} Xe \lhd \neg eq(d, e) \rhd \delta.$$

Using SUM1 and the summand inclusion notation we obtain:

$$\sum_{d:D} Xd \lhd eq(d, e) \rhd \delta \subseteq Xe.$$

$\supseteq$) By applying SUM3, and the assumption that $eq(e, e) = \mathtt{t}$, we find:

$$\sum_{d:D} Xd \lhd eq(d, e) \rhd \delta \supseteq Xe \lhd eq(e, e) \rhd \delta = Xe.$$

$\boxtimes$

**Lemma 2.5** *If there is some $e{:}D$ such that $b(e)$ holds, then*

$$x = \sum_{d:D} x \lhd b(d) \rhd \delta.$$

### 2.2.7 Encapsulation

| | | |
|---|---|---|
| DD | $\partial_H(\delta) = \delta$ | |
| D1 | $\partial_H(a(d)) = a(d)$ | if $a \notin H$ |
| D2 | $\partial_H(a(d)) = \delta$ | if $a \in H$ |
| D3 | $\partial_H(x + y) = \partial_H(x) + \partial_H(y)$ | |
| D4 | $\partial_H(x \cdot y) = \partial_H(x) \cdot \partial_H(y)$ | |
| SUM8 | $\partial_H(\sum X) = \sum_{d:D} \partial_H(Xd)$ | |

Table 6: Axioms for encapsulation.

Sometimes, we want to express that certain actions cannot happen, and must be blocked, i.e. renamed to $\delta$. Generally, this is only done when we want to force this action into a communication. The *encapsulation* operator $\partial_H$ ($H \subseteq \mathsf{Act}$) is specially designed for this task. In $\partial_H(p)$ it prevents all actions of which the action name is mentioned in $H$ from happening. Typically,

$$\partial_{\{b\}}(a \cdot b(3) \cdot c) = a \cdot \delta$$

where $a$, $b$ and $c$ are action names. The properties of $\partial_H$ are axiomatised in Table 6.

### 2.2.8 Internal actions and abstraction

Abstraction is an important means to analyse communicating systems. It means that certain actions are made invisible, so that the relationship between the remaining actions becomes clearer. A specification can be proven equal to an implementation, consisting of a number of parallel processes, after abstracting from all communications between these components.

The *internal action* is denoted by $\tau$. It represents an action that can take place in a system, but that cannot be observed directly. The internal action is meant for analysis purposes, and is hardly ever used in specifications, as it is very uncommon to specify that something unobservable must happen.

Typical identities characterising $\tau$ are $a \cdot \tau \cdot p = a \cdot p$, with $a$ an action and $p$ a process expression. It says that it is impossible to tell by observation whether or not internal actions happen after the $a$. Sometimes, the presence of internal actions can be observed,

due to the context in which they appear. E.g. $a + \tau \cdot b \neq a + b$, as the left hand side can silently execute the $\tau$, after which it only offers a $b$ action, whereas the right hand side can always do an $a$. The difference between the two processes can be observed by insisting in both cases that the $a$ happens. This is always successful in $a + b$, but may lead to a deadlock in $a + \tau \cdot b$.

| | | |
|---|---|---|
| B1 | $c \cdot \tau = c$ | |
| B2 | $x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$ | |
| TID | $\tau_I(\delta) = \delta$ | |
| TIT | $\tau_I(\tau) = \tau$ | |
| TI1 | $\tau_I(a(d)) = a(d)$ | if $a \notin I$ |
| TI2 | $\tau_I(a(d)) = \tau$ | if $a \in I$ |
| TI3 | $\tau_I(x + y) = \tau_I(x) + \tau_I(y)$ | |
| TI4 | $\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$ | |
| SUM9 | $\tau_I(\sum X) = \sum_{d:D} \tau_I(Xd)$ | |
| DT | $\partial_H(\tau) = \tau$ | |

Table 7: Axioms for internal actions and abstraction.

The natural axiom for internal actions is B1 in Table 7. Using the *parallel composition* operator (Section 2.2.9) and encapsulation, B1 can be used to prove all closed instantiations of B2 [vG93], and therefore B2 is also a natural law characterising internal actions. The semantics that is designed around these axioms is rooted branching bisimulation. The axioms in all other tables hold in strong bisimulation semantics, which does not abstract from internal actions. The first semantics abstracting from internal actions is weak bisimulation [Mil80]. Weak bisimulation relates strictly more processes than rooted branching bisimulation, which in turn relates more processes than strong bisimulation. It is a good habit to prove results in the strongest possible semantics, as these results automatically carry over to all weaker variants. We do not consider these semantics explicitly in this section. The reader is referred to for instance [BW90, vG90, Mil89].

In order to abstract from actions, the *abstraction* operator $\tau_I$ ($I \subseteq \mathsf{Act}$) is introduced, where $I$ is a set of action names. The process $\tau_I(p)$ behaves as the process $p$, except that all actions with action names in $I$ are renamed to $\tau$. This is clearly characterised by the axioms in Table 7.

### 2.2.9 Parallel processes

The parallel composition operator can be used to put processes in parallel. The behaviour of $p \parallel q$ is the arbitrary interleaving of actions of the processes $p$ and $q$, assuming for the moment that there is no communication between $p$ and $q$. For example the process $a \parallel b$ behaves like $a \cdot b + b \cdot a$.

The parallel composition operator allows us to describe intricate processes. For instance a bag reading natural numbers using action name $r$ and delivering them via action name $s$ can be described by:

**act** $r, s : Nat$
**proc** $Bag = \sum_{n:Nat} r(n) \cdot (s(n) \parallel Bag)$

Note that the elementary property of bags, namely that at most as many numbers can be delivered as have been received in the past, is satisfied by this description.

It is possible to let processes $p$ and $q$ in $p \parallel q$ communicate. This can be done by declaring in a communication section that certain action names can synchronise. This is done as follows:

**comm** $a \mid b = c$

This means that if actions $a(d_1, \ldots, d_n)$ and $b(d_1, \ldots, d_n)$ can happen in parallel, they may synchronise and this synchronisation is denoted by $c(d_1, \ldots, d_n)$. If two actions synchronise, their arguments must be exactly the same. In a communication declaration it is required that action names $a$, $b$ and $c$ are declared with exactly the same data sorts. It is not necessary that these sorts are unique. It is for example perfectly right to declare the three actions both with a sort $Nat$ and with a pair of sorts $D \times \mathbf{Bool}$.

If a communication is declared as above, synchronisation is another possibility for parallel processes. For example the process $a \parallel b$ is now equivalent to $a \cdot b + b \cdot a + c$. Often, this is not quite what is desired, as the intention generally is that $a$ and $b$ do not happen on their own. For this, the encapsulation operator can be used. The process $\partial_{\{a,b\}}(a \parallel b)$ is equal to $c$.

| | | |
|---|---|---|
| CM1 | $x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x \mid y$ | |
| CM2 | $c \lfloor\!\lfloor x = c \cdot x$ | |
| CM3 | $c \cdot x \lfloor\!\lfloor y = c \cdot (x \parallel y)$ | |
| CM4 | $(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$ | |
| SUM6 | $(\sum X) \lfloor\!\lfloor x = \sum_{d:D} (Xd \lfloor\!\lfloor x)$ | |
| CF | $a(d) \mid a'(e) = \begin{cases} \gamma(a,a')(d) \lhd eq(d,e) \rhd \delta & \text{if } \gamma(a,a') \text{ defined} \\ \delta & \text{otherwise} \end{cases}$ | |
| CD1 | $\delta \mid c = \delta$ | |
| CD2 | $c \mid \delta = \delta$ | |
| CT1 | $\tau \mid c = \delta$ | |
| CT2 | $c \mid \tau = \delta$ | |
| CM5 | $c \cdot x \mid c' = (c \mid c') \cdot x$ | |
| CM6 | $c \mid c' \cdot x = (c \mid c') \cdot x$ | |
| CM7 | $c \cdot x \mid c' \cdot y = (c \mid c') \cdot (x \parallel y)$ | |
| CM8 | $(x + y) \mid z = x \mid z + y \mid z$ | |
| CM9 | $x \mid (y + z) = x \mid y + x \mid z$ | |
| SUM7 | $(\sum X) \mid x = \sum_{d:D} (Xd \mid x)$ | |
| SUM7′ | $x \mid (\sum X) = \sum_{d:D} (x \mid Xd)$ | |

Table 8: Axioms for parallelism in $\mu$CRL.

Axioms that describe the parallel composition operator are in Table 8. In this table the communications between action names from the communication section are represented by the *communication function* $\gamma$. In order to formulate the axioms two auxiliary parallel composition operators have been defined. The left merge $\lfloor\!\lfloor$ is a binary operator that behaves exactly as the parallel composition operator, except that its first action must come from the left hand side. The communication merge $\mid$ is also a binary operator behaving as the parallel composition operator, except that the first action must be a synchronisation between its left and right operand. The core law for the parallel composition operator is CM1 in Table 8. It says that in $x \parallel y$ either $x$ performs the first step, represented by the summand $x \lfloor\!\lfloor y$, or $y$ can do the first step, represented by $y \lfloor\!\lfloor x$, or the first step of $x \parallel y$ is a communication between $x$ and $y$, represented by $x \mid y$. All other axioms in Table 8 are designed to eliminate the parallel composition operators in favour of the alternative composition and the sequential composition operator. The operators for parallel composition ($\parallel$, $\lfloor\!\lfloor$, and $\mid$) bind stronger than the conditional operator and weaker than the sequential composition operator.

Data transfer between parallel components occurs very often. As an example we describe a simplified instance of data transfer. One process sends a natural number $n$ via action name $s$, and another process reads it, via action name $r$ and then announces it via action name $a$. Using an encapsulation and an abstraction operator we force the processes to communicate, and make the communication internal. Of course we expect the process $p$ to be equal to $\tau \cdot a(n)$.

**var** $n{:}Nat$
**act** $r, s, c, a : Nat$
**comm** $r \mid s = c$
**proc** $p = \tau_{\{c\}}(\partial_{\{r,s\}}(s(n) \parallel \sum_{m:Nat} r(m) \cdot a(m)))$

Assuming that $eq$ is an equality function on natural numbers, we have

$$\begin{aligned}
p &= \tau_{\{c\}}(\partial_{\{r,s\}}(s(n) \parallel \textstyle\sum_{m:Nat} r(m) \cdot a(m))) \\
&= \tau_{\{c\}}(\partial_{\{r,s\}}\ (\ s(n) \cdot \textstyle\sum_{m:Nat} r(m) \cdot a(m) \\
&\qquad\qquad +\textstyle\sum_{m:Nat} r(m) \cdot (s(n) \parallel a(m)) \\
&\qquad\qquad +\textstyle\sum_{m:Nat} c(m) \cdot a(m) \lhd eq(n,m) \rhd \delta)) \\
&= \tau_{\{c\}}(\textstyle\sum_{m:Nat} c(m) \cdot a(m) \lhd eq(n,m) \rhd \delta) \\
&= \textstyle\sum_{m:Nat} \tau \cdot a(m) \lhd eq(n,m) \rhd \delta \\
&= \tau \cdot a(n)
\end{aligned}$$

### 2.2.10 Renaming

In some cases it is efficient to reuse a given specification with different action names. This allows, for instance, the definition of generic components that can be used in different configurations. We introduce a *renaming operator* $\rho_R$. The subscript $R$ is a sequence of renamings of the form $a \to b$, meaning that action name $a$ must be replaced by $b$. This sequence of renamings is not allowed to contain distinct entries that replace the same action name. For example the subscript $a \to b, a \to c$ is not allowed. So, clearly, $\rho_R(p)$ is the process $p$ with its action names replaced in accordance with $R$. An equational characterisation of the renaming operator may be found in Table 9.

| | |
|---|---|
| RD | $\rho_R(\delta) = \delta$ |
| RT | $\rho_R(\tau) = \tau$ |
| R1 | $\rho_R(a(d)) = R(a)(d)$ |
| R3 | $\rho_R(x + y) = \rho_R(x) + \rho_R(y)$ |
| R4 | $\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y)$ |
| SUM10 | $\rho_R(\sum X) = \sum_{d:D} \rho_R(Xd)$ |

Table 9: Axioms for renaming in $\mu$CRL.

# 3 A strategy for verification

In process algebra it is common to verify the correctness of a description (the implementation) by proving it equivalent in some sense, e.g. with respect to rooted branching bisimulation, to a more abstract specification. When data is introduced into the descriptions, proving equivalence is more complex, since data can considerably alter the flow of control in the process. The cones and foci technique of [GS95] addresses this problem. A requirement of the cones and foci proof method is that the processes are defined by a linear equation (Definition 3.1). The linearisation of process terms is a common transformation in process algebra. Informally, all operators other than $\cdot$, $+$ and the conditional are eliminated. Therefore, we first present the linear process operator.

## 3.1 Linear process operators

We start out with the definition of 'linear process operator'. The advantage of the linear format is that it is simple. It only uses a few simple process operators in a restricted way. In particular, it does not contain the parallel composition operator. In general a linear process operator can easily be obtained from a $\mu$CRL description, including those containing parallel composition operators, without undue expansion (see also Section 5). Other formats, such as transition systems or automata, generally suffer from exponential blow up when the parallel composition operator is eliminated. This renders them unusable for the analysis of most protocols. We use *linear process operators* and *linear process equations*.

**Definition 3.1** A *linear process operator* (*LPO*) over data sort $D$ is an expression of the

form

$$\lambda p.\lambda d{:}D.\sum_{i\in I}\sum_{e_i:E_i} c_i(f_i(d,e_i))\cdot p(g_i(d,e_i)) \lhd b_i(d,e_i) \rhd \delta$$

for some finite index set $I$, action names $c_i \in \mathsf{Act} \cup \{\tau\}$, data sorts $E_i, D_i$, and functions $f_i : D \times E_i \to D_i$, $g_i : D \times E_i \to D$, and $b_i : D \times E_i \to \mathbf{Bool}$. (We assume that $\tau$ has no parameter.)

Here $D$ represents the state space, $c_i$ are the action names, $f_i$ represents the action parameters, $g_i$ is the state transformation and $b_i$ represents the condition determining whether an action is enabled. Note that the bound variable $p$ ranges over processes parameterised with a datum of sort $D$. We use a meta-sum notation $\sum_{i\in I} p_i$ for $p_1+p_2+\cdots+p_n$ assuming $I = \{1,\ldots,n\}$; the $p_i$'s are called *summands* of $\sum_{i\in I} p_i$. For $I = \emptyset$ we define $\sum_{i\in I} p_i = \delta$. We generally use letters $\Phi$, $\Psi$, and $\Xi$ to refer to LPOs.

According to the definition in [BG94b], an LPO may have summands that allow termination. We have omitted these here, because they hardly occur in actual specifications and obscure the presentation of the theory. Moreover, it is not hard to add them if necessary.

LPOs have been defined as having a single data parameter. Most LPOs that we consider have several parameters, but these may be reduced to one parameter by means of cartesian products and projection functions. Often, parameter lists get rather long. Therefore, we use the following notation for updating elements in the list. Let $\vec{d}$ abbreviate the vector $d_1,\ldots,d_n$. A summand of the form $\sum_{e_i:E_i} c_i(f_i(\vec{d},e_i))\cdot p(d_i'/d_i)\lhd b_i(\vec{d},e_i)\rhd\delta$ in the definition of a process $p(\vec{d})$ abbreviates $\sum_{e_i:E_i} c_i(f_i(\vec{d},e_i))\cdot p(d_1,\ldots,d_{i-1},d_i',d_{i+1},\ldots d_n)\lhd b_i(\vec{d},e_i)\rhd\delta$. Here, the parameter $d_i$ is updated to $d_i'$ in the recursive call. This notation is extended in the natural way to multiple updates. If no parameter is updated, we write the summand as $\sum_{e_i:E_i} c_i(f_i(\vec{d},e_i))\cdot p \lhd b_i(\vec{d},e_i)\rhd\delta$.

Given a process operator $\Psi$, the associated linear process equation (LPE) can be written as $X(d) = \Psi X d$. Conversely, given a linear process equation $X(d) = p$, the associated LPO can be written as $\lambda X.\lambda d{:}D.p$. As a consequence we can choose whether to use linear process operators or equations at each point. Notions defined for LPOs carry over to LPEs in a straightforward manner and vice versa.

As an example consider the unreliable data channel that occurs in the alternating bit protocol [BW90], usually specified by:

$$\textbf{proc}\quad K = \textstyle\sum_{d:D}\sum_{b:Bit} r(\langle d,b\rangle)\cdot(j\cdot s(\langle d,b\rangle) + j'\cdot s_3(ce))\cdot K$$

The channel $K$ reads frames consisting of a datum from some data type $D$ and an alternating bit. It either delivers the frame correctly, or loses or garbles it. In the last case a checksum error $ce$ is sent. The non-deterministic choice between the two options is mod-

eled by the actions $j$ and $j'$. If $j$ is chosen the frame is delivered correctly and if $j'$ happens it is garbled or lost.

The process $K$ can be transformed into linear format by introducing a special variable $i_k$ indicating the state of the process $K$. Just before the $r$ action this state is 1. Directly after it, the state is 2. The state directly after action $j$ is 3, and the state directly after $j'$ is 4. We have indicated these states in the equation by means of encircled numbers:

$$\textbf{proc} \quad K = {}_{①}\textstyle\sum_{d:D}\sum_{b:Bit} r(\langle d,b\rangle)\cdot {}_{②}(j\cdot {}_{③}s(\langle d,b\rangle) + j'\cdot {}_{④}s_3(ce))\cdot {}_{①}K$$

With some experience it is quite easy to see that the channel $K$ has the following linear description:

$$\begin{aligned}\textbf{proc}\quad & K(d{:}D, b{:}Bit, i_k{:}Nat) = \\ & \textstyle\sum_{d':D}\sum_{b':Bit} r(\langle d',b'\rangle)\cdot K(d'/d, b'/b, 2/i_k) \lhd eq(i_k,1)\rhd \delta + \\ & j\cdot K(3/i_k)\lhd eq(i_k,2)\rhd\delta + \\ & j'\cdot K(4/i_k)\lhd eq(i_k,2)\rhd\delta + \\ & s(\langle d,b\rangle)\cdot K(1/i_k)\lhd eq(i_k,3)\rhd\delta + \\ & s(ce)\cdot K(1/i_k)\lhd eq(i_k,4)\rhd\delta\end{aligned}$$

Note that we have deviated from the pure LPO format: in the last four summands there is no summation over the data types $D$ and $Bit$, in the second and third summand $j$ and $j'$ do not carry a parameter and in the first summand there are actually two sum operators. This is easily remedied by introducing dummy summands and dummy arguments, and pairing of variables. Note that linear process equations are not very readable, and therefore, they are less suited for specification purposes.

## 3.2 Proof principles and elementary lemmata

In order to verify recursive processes, we need auxiliary rules. The axioms presented in the previous section are not sufficiently strong to prove typical recursive properties. We introduce here the principles L-RDP (*Linear Recursive Definition Principle*) and CL-RSP (*Convergent Linear Recursive Specification Principle*). All the methods that we present in the sequel are derived from these rules[1].

Processes can be defined as fixed points for convergent LPOs and as solutions for LPEs. In this chapter we use the term *solution* for both.

[1]Elsewhere we also use Koomen's Fair Abstraction Rule, but as we avoid processes with internal loops, we do not need KFAR here.

**Definition 3.2** A *solution* of an LPO $\Phi$ is a process $p$, parameterised with a datum of sort $D$, such that for all $d{:}D$ we have $p(d) = \Phi pd$.

**Definition 3.3** An LPO $\Phi$ written as in Definition 3.1 is called *convergent* iff there is a well-founded ordering $<$ on $D$ such that for all $i \in I$ with $c_i = \tau$ and for all $e_i{:}E_i$, $d{:}D$ we have that $b_i(d, e_i)$ implies $g_i(d, e_i) < d$.

For each LPO $\Phi$, we assume an axiom which postulates that $\Phi$ has a canonical solution. Then, we postulate that every *convergent* LPO has at most one solution. In this way, convergent LPOs define processes. The two principles reflect that we only consider process algebras where every LPO has at least one solution and converging LPOs have precisely one solution.

**Definition 3.4** The *Linear Recursive Definition Principle* (*L-RDP*) says that every linear process operator $\Psi$ has at least one solution, i.e. there exists a $p$ such that for all $d{:}D$ we have $p(d) = \Psi pd$.

The *Convergent Linear Recursive Specification Principle* (*CL-RSP*) [BG94b] says that every convergent linear process operator has at most one solution, i.e. for all $p$ and $q$ if $p = \Psi p$ and $q = \Psi q$, then for all $d{:}D$ we have $p(d) = q(d)$.

The following theorem, proven in [BG94b], says that if an LPO is convergent in the part of its state space that satisfies an invariant $I$, then it has at most one solution in that part of the state space. It has been shown to be equivalent to CL-RSP in [BG94b].

**Definition 3.5** An *invariant* of an LPO $\Phi$ written as in Definition 3.1 is a function $I : D \to \mathbf{Bool}$ such that for all $i \in I$, $e_i{:}E_i$, and $d{:}D$ we have:

$$b_i(d, e_i) \wedge I(d) \to I(g_i(d, e_i)).$$

**Theorem 3.6 (Concrete Invariant Corollary)** *Let $\Phi$ be an LPO. If, for some invariant $I$ of $\Phi$, the LPO $\lambda p.\lambda d.\Phi pd \lhd I(d) \rhd \delta$ is convergent and for some processes $q$, $q'$, parameterised by a datum of sort $D$, we have $I(d) \to q(d) = \Phi qd$ and $I(d) \to q'(d) = \Phi q'd$, then $I(d) \to q(d) = q'(d)$.*

To develop the theory it is convenient to work with a particular form of LPOs, which we call (*action*) *clustered*[2]. Clustered LPOs contain, for each action $a$, at most one summand starting with an $a$. Thus clustered LPOs can be defined by summation over a finite index set $I$ of actions.

[2]At some places clustered LPOs have been called deterministic. However, this is a bad name, as the process underlying a 'deterministic' LPO is not at all a deterministic process, i.e. a process that can for each action $a$ always do at most one $a$ transition.

**Definition 3.7** Let $Act \subseteq \mathsf{Act} \cup \{\tau\}$ be a finite set of action names. A *clustered linear process operator* (*C-LPO*) over $Act$ is an expression of the form

$$\Phi = \lambda p.\lambda d{:}D.\sum_{a \in Act}\sum_{e_a:E_a} a(f_a(d,e_a)) \cdot p(g_a(d,e_a)) \lhd b_a(d,e_a) \rhd \delta.$$

The first part of the following theorem states that it is no restriction to assume that LPOs are clustered. The second part is a prelude on the general equality theorem, as it requires that, for each action, the sorts in the sum operators preceding this action are the same in specification and implementation. A proof is given in [GS95].

**Theorem 3.8**

1. *Every convergent LPO $\Phi$ can be rewritten to a C-LPO $\Phi'$ with the same solution, provided all occurrences of the same action have parameters of the same type.*
2. *Consider convergent C-LPOs $\Phi$, $\Psi$ such that action $a$ occurs both in $\Phi$ and in $\Psi$ (with parameters of the same data type). There exist convergent C-LPOs $\Phi'$, $\Psi'$ having the same solutions as $\Phi$, $\Psi$, respectively, such that $a$ occurs in $\Phi'$ and $\Psi'$ in summands with summation over the same sort $E_a$.*

The two summands $s(\langle d,b\rangle) \cdot K(1/i_k) \lhd eq(i_k,3) \rhd \delta$ and $s(ce) \cdot K(1/i_k) \lhd eq(i_k,4) \rhd \delta$ of the channel $K$ can be grouped together as

$$s(if(eq(i_k,3),\langle d,b\rangle,ce)) \cdot K(1/i_k) \lhd eq(i_k,3) \vee eq(i_k,4) \rhd \delta.$$

Here we assume that $ce$ is of the same sort as the pair $\langle d,b\rangle$.

## 3.3 The general equality theorem

In this section, we are concerned with proving equality of solutions of C-LPOs $\Phi$ and $\Psi$. The C-LPO $\Phi$ defines an implementation and the C-LPO $\Psi$ defines the specification of a system. We use the *cones and foci* proof method of [GS95].

We assume that $\tau$-steps do not occur in the specification $\Psi$. We want to show that after abstraction of internal actions in a set $Int$ the solution of $\Phi$ is equal to the solution of $\Psi$. We assume that $\Phi$ cannot perform an infinite sequence of internal actions. It turns out to be convenient to consider $\Phi$ where the actions in $Int$ are already renamed to $\tau$. Hence, we speak about a C-LPO $\Xi$ which is $\Phi$ where actions in $Int$ have been abstracted from (so $\tau_{Int}(\Phi) = \Xi$). Note that $\Xi$ is convergent, and hence defines a process. We fix the C-LPOs $\Xi$ and $\Psi$ as follows (where the action names are taken from a set $Act$):

$$\Xi = \lambda p.\lambda d{:}D_\Xi.\sum_{a \in Act}\sum_{e_a:E_a} a(f_a(d,e_a)) \cdot p(g_a(d,e_a)) \lhd b_a(d,e_a) \rhd \delta,$$

$$\Psi = \lambda q.\lambda d{:}D_\Psi.\ \sum_{a \in Act \setminus \{\tau\}} \sum_{e_a:E_a} a(f'_a(d,e_a)) \cdot q(g'_a(d,e_a)) \lhd b'_a(d,e_a) \rhd \delta.$$

The issue that we consider is how to prove the solutions of $\Xi$ and $\Psi$ equal.

The main idea of the cones and foci proof method is that there are usually many internal events in the implementation, but they are only significant in that they must progress somehow towards producing a visible event which can be matched with a visible event in the specification. A state of the implementation where no internal actions are enabled is called a *focus point*, and there may be several such points in the implementation. Focus points are characterised by a boolean condition on the data of the process called the *focus condition*. The focus condition $FC_\Xi(d)$ is the negation of the condition which allows $\tau$ actions to occur. The focus condition $FC_\Xi(d)$ is true if $d$ is a focus point and false otherwise.

**Definition 3.9** The *focus condition* $FC_\Xi(d)$ of $\Xi$ is the formula $\neg\exists_{e_\tau:E_\tau}\ (b_\tau(d,e_\tau))$.

The *cone* belonging to a focus point is the part of the state space from which the focus point can be reached by internal actions; imagine the transition system forming a cone or funnel pointing towards the focus. Figure 1 in Section 1 visualises the core observation underlying this method.

The final element in the proof method is a *state mapping* $h : D_\Xi \to D_\Psi$ between the data states of the implementation and the data states of the specification. It explains how the data parameter that encodes states of the specification is constructed out of the data parameter that encodes states of the implementation. This mapping is surjective, but almost certainly not injective, since the data of the specification is likely to be simpler than that of the implementation. So in this respect we have a refinement, but in terms of actions we have an equivalence.

In order to prove implementation and specification rooted branching bisimilar, the state mapping should satisfy certain properties, which we call *matching criteria* because they serve to match states and transitions of implementation and specification. They are inspired by numerous case studies in protocol verification, and reduce complex calculations to a few straightforward checks. If these six criteria are satisfied then the specification and the implementation can be said to be rooted branching bisimilar under the General Equality Theorem (Theorem 3.11). The general forms of the matching criteria are given in Definition 3.10. Given the particular actions, conditions and mapping for a system, the matching criteria can be mechanically derived. Of course, the choice of mapping requires some thought, as does the subsequent proof of the criteria.

Now we formulate the criteria. We discuss each criterion directly after the definition.

**Definition 3.10** Let $h : D_\Xi \to D_\Psi$ be a state mapping. The following criteria referring to $\Xi$, $\Psi$ and $h$ are called the *matching criteria*. We refer to their conjunction by $C_{\Xi,\Psi,h}(d)$.

For all $e_\tau{:}E_\tau$, $a \in Act \setminus \{\tau\}$, and $e_a{:}E_a$:

(1) $\Xi$ is convergent

(2) $b_\tau(d, e_\tau) \to h(d) = h(g_\tau(d, e_\tau))$

(3) $b_a(d, e_a) \to b'_a(h(d), e_a)$

(4) $FC_\Xi(d) \wedge b'_a(h(d), e_a) \to b_a(d, e_a)$

(5) $b_a(d, e_a) \to f_a(d, e_a) = f'_a(h(d), e_a)$

(6) $b_a(d, e_a) \to h(g_a(d, e_a)) = g'_a(h(d), e_a)$

Criterion (1) says that $\Xi$ must be convergent. In effect this does not say anything else than that in a cone every internal action $\tau$ constitutes progress towards a focus point. Criterion (2) says that if in a state $d$ in the implementation an internal step can be done (i.e. $b_\tau(d, e_\tau)$ is valid) then this internal step is not observable. This is described by saying that the state before the $\tau$-step and the state after the $\tau$-step both relate to the same state in the specification. Criterion (3) says that when the implementation can perform an external step, then the corresponding state in the specification must also be able to perform this step. Note that in general, the converse need not hold. If the specification can perform an $a$-action in a certain state $e$, then it is only necessary that in every state $d$ of the implementation such that $h(d) = e$ an $a$-step can be done *after some internal actions*. This is guaranteed by criterion (4). It says that in a focus point of the implementation, an action $a$ in the implementation can be performed if it is enabled in the specification. Criteria (5) and (6) express that corresponding external actions carry the same data parameter (modulo $h$) and lead to corresponding states.

Assume that $r$ and $q$ are solutions of $\Xi$ and $\Psi$, respectively. Using the matching criteria, we would like to prove that, for all $d{:}D$, $C_{\Xi,\Psi,h}(d)$ implies $r(d) = q(h(d))$.

In fact we prove a more complicated result. This has two reasons. The first one is that the statement above is not generally true. Consider the case where $d$ is a non-focus point of $\Xi$. In this case, $r(d)$ can perform a $\tau$-step. Since $q$ cannot perform $\tau$-steps, $r(d)$ cannot be equal to $q(h(d))$. Therefore, in the setting of rooted branching bisimulation we can for non-focus points $d$ only prove $\tau \cdot r(d) = \tau \cdot q(h(d))$.

The second reason why we need a more complicated result is of a very general nature. A specification and an implementation are in general only equivalent for the reachable states in the implementation. A common tool to exclude non-reachable states is an invariant. Therefore we have added an invariant to the theorem below. For a proof of this theorem we refer to [GS95].

**Theorem 3.11 (General Equality Theorem)** *Let $\Xi$ be a C-LPO and let $\Psi$ be a C-LPO that does not contain $\tau$-steps. Let $h$ be a state mapping. Assume that $r$ and $q$ are solutions of $\Xi$ and $\Psi$, respectively. If $I$ is an invariant of $\Xi$ and $I(d) \rightarrow C_{\Xi,\Psi,h}(d)$ for all $d{:}D_\Xi$, then*

$$\forall_{d:D_\Xi}\ (I(d) \rightarrow r(d) \triangleleft FC_\Xi(d) \triangleright \tau \cdot r(d) = q(h(d)) \triangleleft FC_\Xi(d) \triangleright \tau \cdot q(h(d))).$$

## 3.4 Pre-abstraction and idle loops

The proof strategy presented in the previous section can only be applied to systems for which the implementation is convergent. This is an all too serious restriction. In this section we present a generalisation of the proof strategy which is also capable to deal with idle loops.

The most important concept in this generalisation is the *pre-abstraction function*. A pre-abstraction function divides the occurrences of the internal actions into *progressing* and *non-progressing* internal actions. The progressing internal actions are the ones for which the pre-abstraction function gives true and the non-progressing actions are the ones for which the pre-abstraction function gives false.

**Definition 3.12** Let $\Phi$ be a C-LPO and let *Int* be a finite set of action names. A *pre-abstraction function* $\xi$ is a mapping that yields for every action $a \in Int$ an expression of sort $D \times E_a \rightarrow \mathbf{Bool}$. The partial pre-abstraction function $\xi$ is extended to a total function on $\mathsf{Act}$ by assuming $\xi(\tau)(d, e_\tau) = \mathsf{t}$ and $\xi(a)(d, e_a) = \mathsf{f}$ for all $a \in \mathsf{Act} \setminus Int$.

The pre-abstraction function $\xi$ defines from which internal actions we abstract. If the pre-abstraction function of an action yields true (progressing internal action), the action is replaced by $\tau$, while if the pre-abstraction function yields false the action remains unchanged.

In a nutshell, the adaptation of the proof strategy is that instead of renaming all internal actions into $\tau$-actions we only rename the progressing internal actions. As a consequence the notions of convergence and focus point need to be adapted. Instead of considering $\tau$-actions, all internal actions involved in the pre-abstraction must be considered.

**Definition 3.13** Let $\Phi$ be a C-LPO with internal actions *Int* and let $\xi$ be a pre-abstraction function. The C-LPO $\Phi$ is called *convergent with respect to* $\xi$ iff there exists a well-founded ordering $<$ on $D$ such that for all $a \in Int \cup \{\tau\}$, $d{:}D$ and all $e_a{:}E_a$ such that $\xi(a)(d, e_a)$ we have that $b_a(d, e_a)$ implies $g_a(d, e_a) < d$.

**Definition 3.14** Let $\xi$ be a pre-abstraction function. The *focus condition of* $\Phi$ *relative to* $\xi$ is defined by

$$FC_{\Phi,Int,\xi}(d) = \forall_{a \in Int \cup \{\tau\}}\ \forall_{e_a:E_a}\ (\xi(a)(d, e_a) \rightarrow \neg b_a(d, e_a)).$$

**Definition 3.15** Let $\Phi$ be a C-LPO over $Ext \cup Int \cup \{\tau\}$ (pairwise disjoint) and let $\Psi$ be a C-LPO over $Ext$. Let $h : D_\Phi \to D_\Psi$ and let $\xi$ be a pre-abstraction function. The following six criteria are called the *matching criteria for idle loops* and their conjunction is denoted by $CI_{\Phi,\Psi,\xi,h}(d)$. For all $i \in Int \cup \{\tau\}$, $e_i{:}E_i$, $a \in Ext$, and $e_a{:}E_a$:

(1) $\Phi$ is convergent with respect to $\xi$

(2) $b_i(d, e_i) \to h(d) = h(g_i(d, e_i))$

(3) $b_a(d, e_a) \to b'_a(h(d), e_a)$

(4) $FC_{\Xi,Int,\xi}(d) \wedge b'_a(h(d), e_a) \to b_a(d, e_a)$

(5) $b_a(d, e_a) \to f_a(d, e_a) = f'_a(h(d), e_a)$

(6) $b_a(d, e_a) \to h(g_a(d, e_a)) = g'_a(h(d), e_a)$

**Theorem 3.16 (Equality theorem for idle loops)** *Let $\Phi$ be a C-LPO over $Ext \cup Int \cup \{\tau\}$ (pairwise disjoint) and $\Psi$ a C-LPO over $Ext$. Let $h : D_\Phi \to D_\Psi$ and let $\xi$ be a pre-abstraction function. Assume that $r$ and $q$ are solutions of $\Phi$ and $\Psi$ respectively. If $I$ is an invariant of $\Phi$ and $I(d) \to CI_{\Phi,\Psi,\xi,h}(d)$ for all $d{:}D_\Phi$, then*

$$\forall_{d:D_\Phi}\ (I(d) \to \tau \cdot \tau_{Int}(r(d)) = \tau \cdot q(h(d))).$$

A proof of this theorem can be found in [GS95].

# 4 Verification of the Serial Line Interface Protocol

In this section we give a completely worked out example of a simple protocol to illustrate the use of $\mu$CRL and the general equality theorem. The Serial Line Interface Protocol (SLIP) is one of the protocols that is very commonly used to connect individual computers via a modem and a phone line. It allows only one single stream of bidirectional information.

Basically, the SLIP protocol works by sending blocks of data. Each block is a sequence of bytes that ends with the special `end` byte. Confusion can occur when the `end` byte is also part of the ordinary data sequence. In this case, the `end` byte is 'escaped', by placing an `esc` byte in front of the `end` byte. Similarly, to distinguish an ordinary `esc` byte from the escape character `esc`, each `esc` in the data stream is replaced by two `esc` characters. In our modeling of the protocol, we ignore the process of dividing the data into blocks, but only look at the insertion and removal of `esc` characters into the data stream. For simplicity we assume that all occurrences of `end` and `esc` bytes have to be 'escaped'. We model the

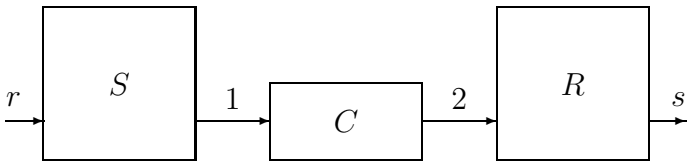



Figure 2: Architecture of the SLIP protocol.

system by three components: a sender ($S$), inserting escape characters, a channel ($C$), modeling the medium along which data is transferred, and a receiver ($R$), removing the escape characters (see Figure 2). We let the channel be a buffer of capacity one in this example.

We use four data types *Nat*, **Bool**, *Byte*, and *Queue* to describe the SLIP protocol and its external behaviour. The sort *Nat* contains the natural numbers. With 0 and $S$ we denote the zero element and the successor function on *Nat* as before. Numerals (e.g. 3) are used as abbreviations. The function $eq : Nat \times Nat \to \mathbf{Bool}$ is true when its arguments represent the same number. The sort **Bool** contains exactly two constants `t` (true) and `f` (false) and we assume that all required boolean connectives are defined in a proper way.

The sort *Byte* contains the data elements to be transferred via the SLIP protocol. As the definition of a byte as a sequence of 8 bits is very detailed and actually irrelevant we only assume about *Byte* that it contains at least two not necessarily different constants `esc` and `end`, and a function $eq : Byte \times Byte \to \mathbf{Bool}$ that represents equality.

**sort** *Byte*
**map** `esc` $:\to Byte$
`end` $:\to Byte$
$eq : Byte \times Byte \to \mathbf{Bool}$

Furthermore, to describe the external behaviour of the system, we use the sort *Queue* as described in Section 2.1, but this time we take the elements of the queue from the sort *Byte*. Additionally, we use the auxiliary function *len* which yields the number of elements in a queue.

**sort** *Queue*
**map** $len : Queue \to Nat$
**var** $b$:*Byte* $q$:*Queue*
**rew** $len([\,]) = 0$
$len(in(b, q)) = S(len(q))$

The processes are defined by guarded process declarations for the channel $C$, the sender $S$ and the receiver $R$ (cf. Figure 2). The equation for the channel $C$ expresses that first a byte $b$ is read using a read action $r_1$ via port 1, and subsequently this value is sent via port 2 using action $s_2$. After this the channel is back in its initial state, ready to receive another byte. The encircled numbers can be ignored for the moment. They serve to explicitly indicate the state of these processes and are used later.

**proc** $C = {}_{⓪}\sum_{b:Byte} r_1(b) \cdot {}_{①}s_2(b) \cdot {}_{⓪}C$

Using the $r$ action the sender reads a byte from a protocol user, who wants to use the service of the SLIP protocol to deliver this byte elsewhere. It is obvious that if $b$ equals `esc` or `end`, first an additional `esc` is sent to the channel (via action $s_1$) before $b$ itself is sent. Otherwise, $b$ is sent without prefix.

**proc** $S = {}_{⓪}\sum_{b:Byte} r(b) \cdot {}_{①}(s_1(\texttt{esc}) \cdot {}_{②}s_1(b) \cdot {}_{⓪}S \lhd eq(b, \texttt{end}) \vee eq(b, \texttt{esc}) \rhd s_1(b) \cdot {}_{⓪}S)$

The receiver is equally straightforward. After receiving a byte $b$ from the channel (via $r_2$) it checks whether it is an `esc`. If so, it removes it and delivers the trailing `end` or `esc`. Otherwise, it just delivers $b$. Both the sender and the receiver repeat themselves indefinitely, too.

**proc** $R = {}_{⓪}\sum_{b:Byte} r_2(b) \cdot {}_{①}((\sum_{b':Byte} r_2(b') \cdot {}_{②}s(b') \cdot {}_{⓪}R) \lhd eq(b, \texttt{esc}) \rhd s(b) \cdot {}_{⓪}R)$

The SLIP protocol is defined by putting the sender, channel and receiver in parallel. We let the actions $r_1$ and $s_1$ communicate and the resulting action is called $c_1$. Similarly, $r_2$ and $s_2$ communicate into $c_2$.

**comm** $s_1 \mid r_1 = c_1$
$s_2 \mid r_2 = c_2$

The encapsulation operator $\partial_{\{r_1,s_1,r_2,s_2\}}$ forbids the actions $r_1$, $s_1$, $r_2$ and $s_2$ to occur on their own by renaming these actions to $\delta$. In this way the actions are forced to communicate. The abstraction operation $\tau_{\{c_1,c_2\}}$ abstracts from these communications by renaming them to the internal action $\tau$. For the SLIP protocol the external actions are $r$ and $s$.

**proc** $SLIP = \tau_{\{c_1,c_2\}}(\partial_{\{r_1,s_1,r_2,s_2\}}(S \parallel C \parallel R))$

We want to obtain a better understanding of the protocol, because although rather simple, it is not straightforward to understand its external behaviour completely. Data that is read at $r$ is of course delivered in sequence at $s$ without loss or duplication of data. So, the protocol behaves like a kind of queue. The reader should now, before reading further, take a few minutes to determine the size of this queue. Simultaneously, one byte can be stored in the receiver, one in the channel and one in the sender. If an `esc` or `end` is in transfer, it matters in which of the processes it is stored. If the `esc` or `end` byte is stored in the sender, no leading `esc` is produced yet. Hence three bytes can be stored in this case. If the `esc` or `end` byte is stored in the channel, it must have been sent there by the sender. Obviously the sender in this case has first sent a leading `esc` byte. This byte is either stored in the receiver or removed by the receiver. In both cases the receiver contains no byte that is visible in the environment. Hence in this case at most two bytes can be stored. Finally, if the `end` or `esc` byte is stored in the receiver, the leading `esc` byte produced by the sender has been removed already by the receiver. Hence three bytes can be stored in this case (assuming that no other `esc` or `end` byte is in transit). So, the conclusion is that the queue behaves as a queue of size three, except when an `esc` or `end` occurs at the second position in the queue (the channel), in which case the size is at most two. For this purpose the auxiliary predicate *full* is defined.

**map** $full : Queue \to \mathbf{Bool}$
**var** $q{:}Queue$
**rew** $full(q) = eq(len(q), 3)$
$\quad \vee (eq(len(q), 2) \wedge (eq(toe(untoe(q)), \texttt{esc}) \vee eq(toe(untoe(q)), \texttt{end})))$

Using this predicate we obtain the description of the external behaviour of the SLIP protocol below: If the queue is not full, an additional byte $b$ can be read. If the queue is not empty an element can be delivered.

**proc** $Spec(q{:}Queue) =$
$\quad \sum_{b:Byte} r(b) \cdot Spec(in(b, q)) \lhd \neg full(q) \rhd \delta +$
$\quad s(toe(q)) \cdot Spec(untoe(q)) \lhd \neg isempty(q) \rhd \delta$

The theorem that we are interested in proving is:

**Theorem 4.1** *We have* $SLIP = Spec([\,])$.

**Proof.** This follows directly from Lemma 4.2 and Lemma 4.4 that are given below. ⊠

We describe the linear equation for *SLIP*. We have numbered the different summands for easy reference. Note that the specification is already linear.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{LinImpl}(b_s{:}\mathit{Byte}, s_s{:}\mathit{Nat}, b_c{:}\mathit{Byte}, s_c{:}\mathit{Nat}, b_r{:}\mathit{Byte}, s_r{:}\mathit{Nat}) = \\
\text{(a)} & \quad \sum_{b:\mathit{Byte}} r(b) \cdot \mathit{LinImpl}(b, 1, b_c, s_c, b_r, s_r) \\
 & \qquad \lhd \mathit{eq}(s_s, 0) \rhd \delta + \\
\text{(b)} & \quad \tau \cdot \mathit{LinImpl}(b_s, 2, \mathtt{esc}, 1, b_r, s_r) \\
 & \qquad \lhd \mathit{eq}(s_c, 0) \wedge \mathit{eq}(s_s, 1) \wedge (\mathit{eq}(b_s, \mathtt{end}) \vee \mathit{eq}(b_s, \mathtt{esc})) \rhd \delta + \\
\text{(c)} & \quad \tau \cdot \mathit{LinImpl}(b_s, 0, b_s, 1, b_r, s_r) \\
 & \qquad \lhd \mathit{eq}(s_c, 0) \wedge (\mathit{eq}(s_s, 2) \vee (\mathit{eq}(s_s, 1) \wedge \neg(\mathit{eq}(b_s, \mathtt{end}) \vee \mathit{eq}(b_s, \mathtt{esc})))) \rhd \delta + \\
\text{(d)} & \quad \tau \cdot \mathit{LinImpl}(b_s, s_s, b_c, 0, b_c, 1) \\
 & \qquad \lhd \mathit{eq}(s_r, 0) \wedge \mathit{eq}(s_c, 1) \rhd \delta + \\
\text{(e)} & \quad \tau \cdot \mathit{LinImpl}(b_s, s_s, b_c, 0, b_c, 2) \\
 & \qquad \lhd \mathit{eq}(s_r, 1) \wedge \mathit{eq}(b_r, \mathtt{esc}) \wedge \mathit{eq}(s_c, 1) \rhd \delta + \\
\text{(f)} & \quad s(b_r) \cdot \mathit{LinImpl}(b_s, s_s, b_c, s_c, b_r, 0) \\
 & \qquad \lhd \mathit{eq}(s_r, 2) \vee (\mathit{eq}(s_r, 1) \wedge \neg \mathit{eq}(b_r, \mathtt{esc})) \rhd \delta
\end{array}
$$

We obtained this form by identifying three explicit states in the sender and receiver, and two in the channel. These have been indicated by encircled numbers in the defining equations of these processes. The states of these processes are indicated by the variables $s_s$, $s_r$ and $s_c$ respectively. Each of the three processes also stores a byte in certain states. The bytes for each process are indicated by $b_s$, $b_r$ and $b_c$. The $\tau$ in summand (b) comes from abstracting from $c_1(\mathtt{esc})$, in summand (c) it comes from $c_1(b_s)$, in (d) from $c_2(b_c)$ and in (e) from $c_2(b_c)$.

The following lemma says that *LinImpl*, the linear equation for *SLIP*, indeed equals the description of *SLIP*.

**Lemma 4.2** *For any $b_1, b_2, b_3{:}\mathit{Byte}$ it holds that*

$$\mathit{LinImpl}(b_1, 0, b_2, 0, b_3, 0) = \mathit{SLIP}.$$

We list below a number of invariants of *LinImpl* that are sufficient to prove the results in the sequel. The proof of the invariants is straightforward, except that we need invariant 2 to prove invariant 3.

**Lemma 4.3** *The following expressions are invariants for LinImpl:*

1. $s_s \leq 2 \wedge s_c \leq 1 \wedge s_r \leq 2$;
2. $\mathit{eq}(s_s, 2) \rightarrow (\mathit{eq}(b_s, \mathtt{esc}) \vee \mathit{eq}(b_s, \mathtt{end}))$;
3. $$
\begin{array}{rl}
\neg \mathit{eq}(s_s, 2) \rightarrow & ((\mathit{eq}(s_c, 0) \wedge \neg(\mathit{eq}(s_r, 1) \wedge \mathit{eq}(b_r, \mathtt{esc})))\vee \\
 & \quad (\mathit{eq}(s_c, 1) \wedge ((\mathit{eq}(s_r, 1) \wedge \mathit{eq}(b_r, \mathtt{esc})) \leftrightarrow \\
 & \qquad (\mathit{eq}(b_c, \mathtt{esc}) \vee \mathit{eq}(b_c, \mathtt{end})))))\wedge \\
\mathit{eq}(s_s, 2) \rightarrow & ((\mathit{eq}(s_c, 1) \wedge \mathit{eq}(b_c, \mathtt{esc}) \wedge \neg(\mathit{eq}(s_r, 1) \wedge \mathit{eq}(b_r, \mathtt{esc})))\vee \\
 & \quad (\mathit{eq}(s_c, 0) \wedge \mathit{eq}(s_r, 1) \wedge \mathit{eq}(b_r, \mathtt{esc}))).
\end{array}
$$

The next step is to relate the implementation and the specification using a state mapping $h : Nat \times Byte \times Nat \times Byte \times Nat \times Byte \to Queue$. For this, we use the auxiliary function *cadd* (conditional add). The expression $cadd(c, b, q)$ yields a queue with byte $b$ added to $q$ if boolean $c$ equals true. If $c$ is false, it yields $q$ itself.

**map** $cadd : \mathbf{Bool} \times Byte \times Queue \to Queue$
**var** $b{:}Byte\ q{:}Queue$
**rew** $cadd(\mathtt{f}, b, q) = q$
$cadd(\mathtt{t}, b, q) = in(b, q)$

The state mapping is in this case:

$$\begin{aligned} &h(b_s, s_s, b_c, s_c, b_r, s_r) = \\ &\quad cadd(\neg eq(s_s, 0), b_s, \\ &\quad cadd(eq(s_c, 1) \wedge (\neg eq(b_c, \mathtt{esc}) \vee (eq(s_r, 1) \wedge eq(b_r, \mathtt{esc}))), b_c, \\ &\quad cadd(eq(s_r, 2) \vee (eq(s_r, 1) \wedge \neg eq(b_r, \mathtt{esc})), b_r, [\,]))). \end{aligned}$$

So, the state mapping constructs a queue out of the state of the implementation, containing at most $b_s$, $b_c$ and $b_r$ in that order. The byte $b_s$ from the sender is in the queue if the sender is not about to read a new byte ($\neg eq(s_s, 0)$). The byte $b_c$ from the channel is in the queue if the channel is actually transferring data ($eq(s_c, 1)$) and if it does not contain an escape character indicating that the next byte must be taken literally. Similarly, the byte $b_r$ from the receiver must be in the queue if it is not empty and $b_r$ is not an escape character.

The focus condition of the SLIP implementation can easily be extracted and is (slightly simplified using the invariant):

$$\begin{aligned} &(eq(s_c, 0) \to eq(s_s, 0)) \wedge \\ &(eq(s_c, 1) \to (\neg eq(s_r, 0) \wedge (eq(s_r, 1) \to \neg eq(b_r, \mathtt{esc})))) \end{aligned}$$

**Lemma 4.4** *For all* $b_1, b_2, b_3{:}Byte$

$$Spec([\,]) = LinImpl(b_1, 0, b_2, 0, b_3, 0).$$

**Proof.** We apply Theorem 3.11 by taking *LinImpl* for $p$, *Spec* for $q$ and the state mapping and invariant provided above. We simplify the conclusion by observing that the invariant and the focus condition are true for $s_s = 0$, $s_c = 0$ and $s_r = 0$. By moreover using that $h(b_1, 0, b_2, 0, b_3, 0) = [\,]$, the lemma is a direct consequence of the generalised equation theorem. We are only left with checking the matching criteria:

1. The measure $13 - s_s - 3s_c - 4s_r$ decreases with each $\tau$ step.
2. (b) Distinction on $s_r$; use invariant. (c) Distinguish different values of $s_s$; use invariant. (d) Trivial. (e) Trivial.

3. (a) Let $\vec{s}$ denote the tuple $(b_s, s_s, b_c, s_c, b_r, s_r)$. We must show that $s_s = 0$ implies $\neg full(h(\vec{s}))$. From $s_s = 0$ and the definitions of $h$ and *full* observe that $full(h(\vec{s}))$ can only be the case if $s_c = 1 \wedge (b_c \neq \texttt{esc} \vee (s_r = 1 \wedge b_r = \texttt{esc})) \wedge (s_r = 2 \vee (s_r = 1 \wedge b_r \neq \texttt{esc}))$ and $b_c = \texttt{esc} \vee b_c = \texttt{end}$. In all other cases we easily find that $\neg full(h(\vec{s}))$. If $b_c = \texttt{esc}$ we find $b_r = \texttt{esc}$ and $s_r = 2$. We also have $s_c = 1$. Using the invariant we obtain $b_c \neq \texttt{esc} \wedge b_c \neq \texttt{end}$. This leads to a contradiction and therefore $\neg full(h(\vec{s}))$. If $b_c = \texttt{end}$ then using hte invariant we find that $s_r = 1$ and $b_r = \texttt{esc}$. This contradicts the above assumption. Therefore also in this case $\neg full(h(\vec{s}))$. (f) Trivial.

4. (a) Let $\vec{s}$ denote the tuple $(b_s, s_s, b_c, s_c, b_r, s_r)$. We must show that if the focus condition and $\neg full(h(\vec{s}))$ hold, then $eq(s_s, 0)$. The proof proceeds by deriving a contradiction under the assumption $\neg eq(s_s, 0)$. If $eq(s_s, 1)$ it follows from the invariant and the focus condition that $len(h(\vec{s})) = 3$, contradicting that $\neg full(h(\vec{s}))$. If $eq(s_s, 2)$, then $len(h(\vec{s})) = 2$, $toe(untoe(h(\vec{s}))) = b_s$ and $eq(b_s, \texttt{esc}) \vee eq(b_s, \texttt{end})$ in a similar way. Also in this case this contradicts $\neg full(h(\vec{s}))$.

   (f) Let $\vec{s}$ denote the tuple $(b_s, s_s, b_c, s_c, b_r, s_r)$. We must show that the invariant, the focus condition and the statement $\neg isempty(h(\vec{s}))$ imply $eq(s_r, 2) \vee (eq(s_r, 1) \wedge \neg eq(b_r, \texttt{esc}))$. Assume FC and, towards using contraposition, $\neg eq(s_r, 2) \wedge (\neg eq(s_r, 1) \vee eq(b_r, \texttt{esc}))$. Using the invariant we deduce $eq(s_r, 0) \vee (eq(b_r, \texttt{esc}) \wedge eq(s_r, 1)$. By the second conjunct of the FC (contraposition), we obtain $\neg eq(s_c, 1)$, so by the invariant, $eq(s_c, 0)$, and by the first conjunct of FC, $eq(s_s, 0)$ holds. By the definition of the state mapping $h$, we easily see that $h(\vec{s}) = [\,]$.

5. (a) Trivial. (f) Use $toe(cadd(c_1, b_1, cadd(c_2, b_2, in(b_3, [\,]))))) = b_3$.

6. (a) Trivial using definitions. (f) Idem.

$\boxtimes$

# 5 Calculating with $n + 1$ similar processes

## 5.1 Introduction

Distributed algorithms are generally configured as an arbitrarily large but finite set of processors that run similar programs. Using $\mu$CRL this can be neatly described. Assume that the individual processes are given by $P(k)$, where $k$:*Nat* is the index of the process. The following equation puts $n + 1$ of these processes in parallel:

$$Sys(n{:}Nat) = P(0) \lhd eq(n, 0) \rhd (Sys(n - 1) \parallel P(n)). \tag{2}$$

Clearly, the process $Sys(n)$ stands for $P(0) \parallel P(1) \parallel \ldots \parallel P(n)$.

We find descriptions along this line in verifications of the bakery protocol [GK94], Milner's scheduler [KS94], a leader election protocol [FGK97], grid protocols [BHP97], and a summing protocol [GMS97].

The description in equation (2) gives rise to two issues. The first one is whether the equation *defines* in a proper way that *Sys* is the parallel composition of the processes $P(k)$. It is clear that the parallel composition of processes $P(k)$ is a solution for *Sys*. In this section we show that, assuming the principle CL-RSP (see Definition 3.4), this is the only solution for *Sys* in (2). So, an equation in the form of (2) is indeed a proper definition.

The second issue is to extract a linear process equation from the description in (2). We show in general terms how given a linear format for the processes $P(k)$, a process in linear format equivalent to $Sys(n)$ can be given.

## 5.2 Linearisation of two different parallel processes

To provide the reader of this section with a basic understanding of the issues involved in the linearisation of the parallel composition of processes, we provide a simple example first. Two LPEs will be composed and a linear process equation for their parallel composition will be given.

We provide here the linearisation of the parallel composition of two linear processes $P(d)$ and $P'(d')$. The parameters $d$ and $d'$ of certain arbitrary sorts $D$ and $D'$ denote the parameters of the LPEs. We assume that the processes are defined by linear equations of the following form:

$$P(d{:}D) = \sum_{i\in I} \sum_{e_i:E_i} a_i(f_i(d,e_i)) \cdot P(g_i(d,e_i)) \lhd c_i(d,e_i) \rhd \delta,$$

and

$$P'(d'{:}D') = \sum_{i'\in I'} \sum_{e'_{i'}:E'_{i'}} a'_{i'}(f'_{i'}(d',e'_{i'})) \cdot P'(g'_{i'}(d',e'_{i'})) \lhd c'_{i'}(d',e'_{i'}) \rhd \delta.$$

We also assume that these equations are convergent, since by CL-RSP this guarantees that they define unique processes.

Now consider the LPE $Q(d,d')$ defined by the following equation:

$$\begin{array}{ll} & Q(d{:}D, d'{:}D') \\ = & \sum_{i\in I} \sum_{e_i:E_i} a_i(f_i(d,e_i)) \cdot Q(g_i(d,e_i),d') \lhd c_i(d,e_i) \rhd \delta \\ + & \sum_{i'\in I'} \sum_{e'_{i'}:E'_{i'}} a'_{i'}(f'_{i'}(d',e'_{i'})) \cdot Q(d,g'_{i'}(d',e'_{i'})) \lhd c'_{i'}(d',e'_{i'}) \rhd \delta \\ + & \sum_{i\in I} \sum_{i'\in I'} \sum_{e_i:E_i} \sum_{e'_{i'}:E'_{i'}} \gamma(a_i(f_i(d,e_i)), a'_{i'}(f'_{i'}(d',e'_{i'}))) \cdot Q(g_i(d,e_i), g'_{i'}(d',e'_{i'})) \\ & \lhd c_i(d,e_i) \wedge c'_{i'}(d',e'_{i'}) \rhd \delta. \end{array}$$

The first summand describes the cases that an action from $P(d)$ can be executed. The second summand describes the cases that an action from $P'(d')$ can be executed. The third summand describes the cases that the two processes try to communicate. In each summand the change of the state is only due to the original processes involved in the action that is executed.

Although the process $Q$ is strictly speaking not an LPE, it is obvious that an LPE $Q'$ can be given that is equivalent. This is left as an exercise to the reader.

**Theorem 5.1** *Let $P$, $P'$, and $Q$ be the LPEs given above. For all $d{:}D$ and $d'{:}D'$ we have*

$$P(d) \parallel P'(d') = Q(d, d').$$

## 5.3 Linearisation of parallel processes

We shall now describe the linearisation of $n+1$ parallel linear processes $P(k, d)$. The natural number $k$ $(0 \leq k \leq n)$ is the index of the process and the parameter $d$ of some arbitrary sort $D$ denotes the other parameters. We assume that each process $P(k, d)$ is defined using a linear equation of the form:

$$P(k{:}Nat, d{:}D) = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(k, d, e_i)) \cdot P(k, g_i(k, d, e_i)) \lhd c_i(k, d, e_i) \rhd \delta. \tag{3}$$

We also assume that this equation is convergent, as this guarantees that this equation defines a unique process.

In order to define the parallel composition we need to determine the 'state space' of the LPE $Q$. The state space of $Q$ is built from the state spaces of the LPEs $P(k, d)$ that are composed in parallel. As we do not know in advance the number of processes composed (i.e. $n$) we define a new sort *DTable*, which is a table indexed by natural numbers containing elements of the sort $D$. In this table the $k^{\text{th}}$ entry represents the state space of process $P(k, d)$. In order to do so, we also need an auxiliary function $\mathit{if} : \mathbf{Bool} \times D \times D \to D$ for $\mathit{if} - \mathit{then} - \mathit{else}$.

**map** $\mathit{if} : \mathbf{Bool} \times D \times D \to D$
**var** $d, d'{:}D$
**rew** $\mathit{if}(\mathsf{t}, d, d') = d$
$\mathit{if}(\mathsf{f}, d, d') = d'$

In the sequel we assume an equality function $eq : D \times D \to \mathbf{Bool}$.

The constant *emT* of sort *DTable* is the empty table. The function *upd* places a new entry with an index in the table and the function *get* gets a specific entry from the table using its

index. We characterise these operators by one single equation. Note that we do not specify what happens if an element is read from the empty table. We refer to the characterising axiom for tables as the *table axiom*. Besides this axiom, we use the fact that tables are generated from the empty table by the update function. This allows us the use of induction on these two operations.

**sort** $DTable$
**func** $emT :\rightarrow DTable$
$upd : Nat \times D \times DTable \rightarrow DTable$
**map** $get : Nat \times DTable \rightarrow D$
**var** $n, m{:}Nat$
$d{:}D$
$dt{:}DTable$
**rew** $get(n, upd(m, d, dt)) = if(eq(n, m), d, get(n, dt))$

In the remainder we write $dt[i]$ instead of $get(i, dt)$.

We can use the following process definition to put $n$ of the processes $P(k, d)$ in parallel.

$$Sys(n{:}Nat, dt{:}DTable) = P(0, dt[0]) \lhd eq(n, 0) \rhd (P(n, dt[n]) \parallel Sys(n-1, dt)). \qquad (4)$$

We assume that there is a commutative and associative communication function $\gamma$ that explains how two actions can synchronise. In case two actions do not synchronise it yields $\delta$. In this section we assume the so-called handshaking axiom, that says that no more than two actions can synchronise. In other words, for all actions $a_1$, $a_2$ and $a_3$, $\gamma(a_1, \gamma(a_2, a_3)) = \delta$ (cf. [BW90]).

In this section we work towards a linear description of $Sys(n, dt)$ (Lemma 5.3). As a bonus we get that $Sys(n, dt)$ has at most one solution (Corollary 5.5). We also provide an alternative transformed linear description which we believe to be more convenient in concrete instances (Theorem 5.6).

The following lemma will be useful in the calculations in the sequel.

**Lemma 5.2** *For $c_1, c_2{:}$**Bool**, $d, d_1, d_2, d_3{:}D$, $m, m_1, m_2, n{:}Nat$, and $dt{:}DTable$:*

1. $\neg(c_1 \wedge c_2) \rightarrow if(c_1, d_1, if(c_2, d_2, d_3)) = if(c_2, d_2, if(c_1, d_1, d_3))$;
2. $m > n \rightarrow Sys(n, dt) = Sys(n, upd(m, d, dt))$;
3. $m_1 \neq m_2 \rightarrow upd(m_1, d_1, upd(m_2, d_2, dt))[n] = upd(m_2, d_2, upd(m_1, d_1, dt))[n]$.

**Proof.** The first two facts are straightforwardly proven by induction on $c_1$ and $n$, respectively. The last fact follows directly by the table axiom and the first item of this lemma. $\boxtimes$

Below we present the main lemma of this section. It gives an expansion of *Sys*. As has been stated above, we find the form of this expansion not very convenient as it has the condition $k_1 > k_2$ in its second group of summands. The more convenient form in Theorem 5.6 restricts the number of alternatives by requiring $i_2 \leq i_1$. However, contrary to the linearisation in Theorem 5.6 we can prove this lemma by induction on $n$.

**Lemma 5.3** *The process Sys as defined in equations (3) and (4) is a solution for Q in equation (5) below.*

$$
\begin{aligned}
&Q(n{:}Nat, dt{:}DTable) = \\
&\quad \textstyle\sum_{i\in I} \sum_{k:Nat} \sum_{e_i:E_i} a_i(f_i(k, dt[k], e_i)) \cdot Q(n, upd(k, g_i(k, dt[k], e_i), dt)) \\
&\qquad \lhd c_i(k, dt[k], e_i) \wedge k \leq n \rhd \delta + \\
&\quad \textstyle\sum_{i_1\in I} \sum_{i_2\in I} \sum_{k_1:Nat} \sum_{k_2:Nat} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, dt[k_1], e_{i_1}))\cdot \qquad (5) \\
&\qquad Q(n, upd(k_1, g_{i_1}(k_1, dt[k_1], e_{i_1}), upd(k_2, g_{i_2}(k_2, dt[k_2], e_{i_2}), dt))) \\
&\qquad\quad \lhd c_{i_1}(k_1, dt[k_1], e_{i_1}) \wedge c_{i_2}(k_2, dt[k_2], e_{i_2}) \wedge \\
&\qquad\qquad eq(f_{i_1}(k_1, dt[k_1], e_{i_1}), f_{i_2}(k_2, dt[k_2], e_{i_2})) \wedge k_1 > k_2 \wedge k_1 \leq n \rhd \delta.
\end{aligned}
$$

**Lemma 5.4** *Equation (5) is convergent.*

**Proof.** As (3) is convergent, there is a well-founded relation $<$ on $Nat \times D$, such that if $c_i(k, d, e_i)$ and $a_i = \tau$, then $(k, g_i(k, d, e_i)) < (k, d)$.

Using $<$ we can define a well-founded relation $\prec$ on $Nat \times DTable$ as follows:

$$
\begin{array}{lll}
(n_1, dt_1) \prec (n_2, dt_2) & \text{iff} & eq(n_1, n_2) \\
& \wedge & \text{for all } 0 \leq k \leq n_1 : (k, dt_1[k]) \leq (k, dt_2[k]) \\
& \wedge & \text{for some } 0 \leq k \leq n_1 : (k, dt_1[k]) < (k, dt_2[k])
\end{array}
$$

where $(k_1, d_1) \leq (k_2, d_2)$ iff $(k_1, d_1) < (k_2, d_2)$, or $eq(k_1, k_2)$ and $eq(d_1, d_2)$.

It is easy to see that $\prec$ is a convergence witness for equation (5). ⊠

**Corollary 5.5** *Equation (4) has at most one solution for the variable Sys.*

**Proof.** Lemma 5.3 says that any solution for *Sys* in (4) is a solution for $Q$ in (5). By Lemma 5.4 and CL-RSP there is at most one solution for $Q$ in (5). Henceforth, equation (4) has at most one solution, too. ⊠

We consider the following theorem the main result of this section, as it provides a linear equivalent of equation (4) that is easy to use and to obtain. We assume that there is a total reflexive ordering $\leq$ on $I$. As $I$ is an index set, this is a very reasonable assumption.

**Theorem 5.6** *The process Sys as defined in equations (3) and (4) is the (unique) solution for Q in the (convergent) equation below; so* $Sys(n, dt) = Q(n, dt)$ *for all* $n{:}Nat$ *and* $dt{:}DTable$.

$$
\begin{aligned}
&Q(n{:}Nat, dt{:}DTable) = \\
&\quad \textstyle\sum_{i\in I} \sum_{k:Nat} \sum_{e_i:E_i} a_i(f_i(k, dt[k], e_i))\, Q(n, upd(k, g_i(k, dt[k], e_i), dt)) \\
&\qquad \lhd c_i(k, dt[k], e_i) \wedge k \leq n \rhd \delta + \\
&\quad \textstyle\sum_{i_1\in I} \sum_{i_2\in I \wedge i_2 \leq i_1} \sum_{k_1:Nat} \sum_{k_2:Nat} \sum_{e_{i_1}:E_{i_1}} \sum_{e_{i_2}:E_{i_2}} \gamma(a_{i_1}, a_{i_2})(f_{i_1}(k_1, dt[k_1], e_{i_1})) \\
&\qquad Q(n, upd(k_1, g_{i_1}(k_1, dt[k_1], e_{i_1}), upd(k_2, g_{i_2}(k_2, dt[k_2], e_{i_2}), dt))) \\
&\qquad\quad \lhd c_{i_1}(k_1, dt[k_1], e_{i_1}) \wedge c_{i_2}(k_2, dt[k_2], e_{i_2}) \wedge \\
&\qquad\quad\; eq(f_{i_1}(k_1, dt[k_1], e_{i_1}), f_{i_2}(k_2, dt[k_2], e_{i_2})) \wedge \\
&\qquad\quad\; \neg eq(k_1, k_2) \wedge k_1 \leq n \wedge k_2 \leq n \rhd \delta.
\end{aligned}
$$

# 6 The Tree Identify Protocol of IEEE 1394 in $\mu$CRL

We apply the cones and foci technique (Section 3.3) and the linearisation of a number of similar parallel processes (Section 5) to a fragment of the software for a high performance serial multimedia bus, the IEEE standard 1394 [IEE96], also known as "Firewire".

Briefly, IEEE 1394 connects together a collection of systems and devices in order to transport all forms of digitalised video and audio quickly, reliably, and inexpensively. Its architecture is scalable, and it is "hot-pluggable", so a designer or user can add or remove systems and peripherals easily at any time. The only requirement is that the form of the network should be a tree (other configurations lead to errors).

The protocol is subdivided into layers, in the manner of OSI, and further into phases, corresponding to particular tasks, e.g. data transmission or bus master identification. Various parts of the standard have been verified using various formalisms and proof techniques. For example, the operation of sending packets of information across the network is described using $\mu$CRL in [Lut97] and shown to be faulty using E-LOTOS in [SM97]. The former is essentially a description only, with five correctness properties stated informally, but not formalised or proved. The exercise of [SM97] is based on the $\mu$CRL description, adding another layer of the protocol and carrying out the verification as suggested, using the tool CADP [FGK$^+$96].

In this section we concentrate on the tree identify phase of the physical layer which occurs after a bus reset in the system, e.g. when a node is added to or removed from the network. The purpose of the tree identify protocol is to assign a (new) root, or leader, to the network. Essentially, the protocol consists of a set of negotiations between nodes to establish the direction of the parent-child relationship. Potentially, a node can be a parent to many nodes, but a child of at most one node. A node with no parent (after the negotiations are complete) is the leader. The tree identify protocol must ensure that a leader is chosen, and

that it is the only leader chosen.

The specification of the external behaviour of the protocol merely announces that a single leader has been chosen. In the implementation, nodes are specified individually and negotiate with their neighbours to determine the parent-child relationship. Communication is by handshaking. These descriptions may be found in Section 6.1. They were derived with reference to the transition diagram in Section 4.4.2.2 of the standard [IEE96].

We prove, using the cones and foci technique, that the implementation has the same behaviour with respect to rooted branching bisimulation as the specification, thereby showing that the implementation behaves as required, i.e. a single leader is chosen. The proofs may be found in Section 6.2.

Several papers deal with the formal description and analysis of parts of the P1394 protocol. See e.g. [DGRV97, Lut97, SM97, SvdZ98].

## 6.1 Description of the Tree Identify Protocol

The $\mu$CRL data definitions used here (e.g. *Nat*, *NatSet*, *NatSetList*) are assumed and not presented; they are straightforward and examples of these or similar types may be found in [Lut97].

The most abstract specification of the tree identify protocol is the one which merely reports that a leader has been found. The network is viewed as a whole, and no communications between nodes are specified. We define

$$Spec = leader \cdot \delta.$$

In the description of the implementation each node in the network is represented by a separate process. Individual nodes are specified below as processes *Node*. Nodes are described by three parameters:

- a natural number $i$: the identification number of the node. This is used to parameterise communications between nodes, and is not changed during any run of the protocol;
- a set of natural numbers $p$: the set of node identifiers of potential parents of the node. The initial value is the set of all neighbours, decreasing to either a singleton (containing the parent node) or the empty set (indicating that the node is the elected leader);
- a natural number $s$: the current state of the node. We use two state values: 0 corresponds to "still working" and 1 to "finished". The initial value is 0.

The identification number of nodes has been introduced to aid specification and does not appear in [IEE96]. In reality a device has a number of ports and knows whether or not a port is connected to another node; there is no need for node identifiers.

A node can send and receive messages: an action $s(i,j)$ is the sending of a request by node $i$ to node $j$ to become its parent, and an action $r(i,j)$ is the receiving of a parent request from node $i$ by node $j$. When the nodes of the network are composed in parallel, these two actions synchronise with each other to produce a $c$ action. An action $c(i,j)$ is the establishment of a child-parent relation between node $i$ and node $j$, where $i$ is the child and $j$ is the parent.

We define the actions and the communications by the following $\mu$CRL specification:

**act** $s, r, c : Nat \times Nat$
$leader$
**comm** $s \mid r = c$

If a node is still active and its set of potential parents is empty, it declares itself leader by the execution of the *leader* action. By definition, nodes in state 1 are equivalent to deadlock. An individual node with identification number $i$ is defined by means of the process $Node(i,p,s)$. The processes $Node(i,p,s)$ are defined by the following LPE.

**Definition 6.1 (Implementation of a node)**

$$\begin{array}{ll} & Node(i{:}Nat, p{:}NatSet, s{:}Nat) \\ = & leader \cdot Node(i,p,1) \lhd s = 0 \wedge isempty(p) \rhd \delta \\ + & \sum_{j:Nat} r(j,i) \cdot Node(i, p \setminus \{j\}, s) \lhd s = 0 \wedge j \in p \rhd \delta \\ + & \sum_{j:Nat} s(i,j) \cdot Node(i,p,1) \lhd s = 0 \wedge p = \{j\} \rhd \delta. \end{array}$$

The implementation then consists of the parallel composition of $n+1$ nodes where the loose send and receive actions are encapsulated: $H = \{s, r\}$. This implementation is described by the process $Imp(n, P_0)$, with $P_0$ describing the configuration of the network:

$$Imp(n{:}Nat, P_0{:}NatSetList) = \partial_H(Nodes(n, P_0)),$$

where

$$Nodes(n, P_0) = Node(0, P_0[0], 0) \lhd n = 0 \rhd (Node(n, P_0[n], 0) \parallel Nodes(n-1, P_0)).$$

$P_0$ is a list of sets of connections for all nodes indexed by node number; it gives the initial values for the sets of potential parents. Initially all nodes are in state 0.

## 6.2 Correctness of the implementation

As mentioned earlier, the protocol operates correctly only on tree networks, i.e. under the assumption of a good network topology. Networks with loops will cause a timeout in the real protocol, and unconnected nodes will simply be regarded as another network. The property of *GoodTopology* is formalised below.

**Definition 6.2** Given $n$:*Nat*, the maximal node identifier in the network, and a list $P_0$:*NatSetList* giving a set of neighbours for all nodes in the network, the conjunction of the following properties is called $\mathit{GoodTopology}(n, P_0)$:

- $P_0$ is symmetric: $\forall_{i,j}\ (i \in P_0[j] \leftrightarrow j \in P_0[i])$.
- $P_0$ is a tree, i.e. it is a connected graph with no loops.

As a preliminary step to applying the cones and foci proof method, the process *Spec* defined in Section 6.1 must be translated into linear form. In order to do so, a data parameter must be added on which to base a state mapping from the data of process *Imp*. We define

$$\mathit{L\text{-}Spec}(b{:}\mathbf{Bool}) = \mathit{leader} \cdot \mathit{L\text{-}Spec}(\mathsf{f}) \lhd b \rhd \delta.$$

Then, the process $\mathit{L\text{-}Spec}(\mathsf{t})$ and the original specification *Spec* are equivalent.

**Theorem 6.3** *Let Spec and L-Spec be as above. Then* $\mathit{L\text{-}Spec}(\mathsf{t}) = \mathit{Spec}$.

**Proof.** The following computation clearly establishes the equivalence: $\mathit{L\text{-}Spec}(\mathsf{t}) = \mathit{leader} \cdot \mathit{L\text{-}Spec}(\mathsf{f}) \lhd \mathsf{t} \rhd \delta = \mathit{leader} \cdot \mathit{L\text{-}Spec}(\mathsf{f}) = \mathit{leader} \cdot (\mathit{leader} \cdot \mathit{L\text{-}Spec}(\mathsf{f}) \lhd \mathsf{f} \rhd \delta) = \mathit{leader} \cdot \delta = \mathit{Spec}$. ⊠

The linearisation of *Imp* is given by the following LPE for *L-Imp*.

$$\begin{array}{ll} & \mathit{L\text{-}Imp}(n{:}\mathit{Nat}, P{:}\mathit{NatSetList}, S{:}\mathit{NatList}) \\ = & \sum_{i:\mathit{Nat}}\ \mathit{leader} \cdot \mathit{L\text{-}Imp}(1/S[i]) \lhd S[i] = 0 \wedge \mathit{isempty}(P[i]) \wedge i \leq n \rhd \delta \\ + & \sum_{i,j:\mathit{Nat}} c(j,i) \cdot \mathit{L\text{-}Imp}((P[i] \setminus \{j\})/P[i], 1/S[j]) \\ & \quad \lhd S[j] = 0 \wedge P[j] = \{i\} \wedge S[i] = 0 \wedge j \in P[i] \wedge i \neq j \wedge i, j \leq n \rhd \delta \end{array}$$

This linearisation can be derived straightforwardly from the definition of individual nodes using the linearisation technique of Section 5.

**Theorem 6.4** *Let* $S_0$:*NatList be the list of initial state values for the nodes, so for all* $i$:*Nat we have* $S_0[i] = 0$. *Then* $\mathit{Imp}(n, P_0) = \mathit{L\text{-}Imp}(n, P_0, S_0)$.

The proof of correctness also requires an invariant on the data states of the implementation. The invariant $I(n, P, S)$ is the conjunction of the invariants listed below. These invariants hold in every state that can be reached from the initial state $(n, P_0, S_0)$. The variables $i$ and $j$ are universally quantified over $\{0, \ldots, n\}$. The notation $singleton(X)$ represents the predicate $|\,X\,| = 1$, i.e. it expresses that the set $X$ contains precisely one element.

$$\mathrm{I}_1: \quad S[i] = 0 \vee S[i] = 1$$

$$\mathrm{I}_2: \quad j \in P_0[i] \leftrightarrow j \in P[i] \vee i \in P[j]$$

$$\mathrm{I}_3: \quad j \in P_0[i] \wedge j \notin P[i] \rightarrow S[j] = 1$$

$$\mathrm{I}_4: \quad S[i] = 1 \rightarrow singleton(P[i]) \vee isempty(P[i])$$

$$\mathrm{I}_5: \quad j \in P[i] \wedge S[i] = 0 \rightarrow S[j] = 0 \wedge i \in P[j]$$

The proofs of these invariants are straightforward, and omitted here. Invariant $\mathrm{I}_1$ expresses that each of the components is in state 0 or in state 1. Invariant $\mathrm{I}_2$ expresses that if the nodes $i$ and $j$ are connected initially, then at all times one is a potential parent of the other. This means that no connections are removed. Invariant $\mathrm{I}_3$ expresses that the potential parent relationship between a node $j$ and a node $i$ is only destroyed if $i$ becomes the parent node of $j$. Invariant $\mathrm{I}_4$ states that if a node is done there is at most one potential parent left. Invariant $\mathrm{I}_5$ expresses that if a node is still busy and has another node as its potential parent then also this potential parent is still busy and considers the other node as a potential parent.

The linearisation of *L-Imp* is not sufficient to allow us to apply Theorem 3.11. A prerequisite for applying the cones and foci technique is that the indices of the alternative quantifications preceding a visible action must be the same in the specification and the implementation; clearly this is not the case. The summation over the node identifiers preceding the *leader* action in *L-Imp* correctly reflects that any node can be the root, i.e. there are multiple foci. However, it is not important *which* node is the root, only that one is chosen, and the boolean condition guarding the *leader* action in *L-Imp* ensures that this is the case. We adapt the specification in such a way that the *leader* action there is also preceded by an alternative quantification over the node identifiers. Clearly the linear specification obtained in this way is equal to the old linear specification.

$$\textit{L-Spec}(b{:}\mathbf{Bool}) = \sum_{i:Nat} leader \cdot \textit{L-Spec}(\mathsf{f}) \lhd b \wedge i \leq n \rhd \delta.$$

The theorem to be demonstrated can now be stated as:

**Theorem 6.5** *Under the assumption of* $GoodTopology(n, P_0)$ *and* $I(n, P_0, S_0)$,

$$\tau \cdot \textit{L-Spec}(\mathsf{t}) = \tau \cdot \tau_{\{c\}}(\textit{L-Imp}(n, P_0, S_0)).$$

*In the special case where $n = 0$ (there is only one node in the network) we have*

$$\mathit{L\text{-}Spec}(\mathtt{t}) = \tau_{\{c\}}(\mathit{L\text{-}Imp}(n, P_0, S_0)).$$

This is a direct instantiation of Theorem 3.11 with the initial state, because in the initial state the focus condition (defined below) is true if and only if $n = 0$. In order to prove Theorem 6.5 the matching criteria must be satisfied. To show that the matching criteria hold we first define the focus condition and the state mapping for $\tau_{\{c\}}(\mathit{L\text{-}Imp})$. The focus condition $FC$ is the condition under which no more $\tau$ steps can be made, i.e. it is the negation of the condition for making a $\tau$ step:

$$FC(n, P, S) = \forall_{i,j \leq n}\ (S[i] = 1 \vee P[i] \neq \{j\} \vee S[j] = 1 \vee i \notin P[j] \vee\ i = j)$$

The state mapping $h$ is a function mapping data states of the implementation into data states of the simple specification. In this case $h$ is defined so that it is $\mathtt{t}$ before the visible *leader* action occurs and $\mathtt{f}$ afterwards:

$$h(n, P, S) = \neg(\forall_{i \leq n}(S[i] = 1)).$$

If a node can do the leader action then all other nodes are in state 1. So, if a node declares itself the leader, then it is the first to do so, and because after this all nodes are in state 1, there will be no subsequent leader action.

**Lemma 6.6 (Uniqueness of Root)**

$$\forall_{i \leq n}\ (\mathit{isempty}(P[i])) \rightarrow \forall_{j \leq n}\ (j \neq i \rightarrow S[j] = 1)$$

**Proof.** We assume nodes $i, j \leq n$ such that $i \neq j \wedge \mathit{isempty}(P[i]) \wedge S[j] = 0$, and derive a contradiction. By *GoodTopology* there is a path of distinct nodes $i = i_0 \ldots i_m = j$, such that $\forall_{k<m}\ (i_{k+1} \in P_0[i_k])$. By $\mathrm{I}_2$ and $\mathit{isempty}(P[i_0])$ we see that $i_0 \in P[i_1]$. Then by $\mathrm{I}_3$ $S[i_1] = 1$, and by $\mathrm{I}_4$ $\mathit{singleton}(P[i_1])$. In a similar way we derive for all $0 < k \leq m$ that $P[i_k] = \{i_{k-1}\}$ and $S[i_k] = 1$. So in particular $S[j] = 1$. ⊠

**The matching criteria** Given the particulars of *L-Imp*, *L-Spec*, *FC* and $h$, the matching criteria may be mechanically derived from the general forms of Definition 3.10. The instantiated matching criteria are stated below, together with the proofs that they hold.

1. The implementation is convergent. Each $\tau$ step decreases the number of nodes $i$ for which $S[i] = 0$ by one.

2. In any data state $d = (n, P, S)$ of the implementation, the execution of an internal step leads to a state with the same $h$-image.

   Suppose an internal action is possible, i.e. there are nodes $i, j \leq n$ such that

   $$S[i] = 0 \wedge P[i] = \{j\} \wedge S[j] = 0 \wedge i \in P[j] \wedge \ i \neq j.$$

   We see that $S[i] = 0$ and $S[j] = 0$ and hence $h(d) = \mathsf{t}$. We have to show that if we reach a state $d' = (n, P', S')$ by the communication between nodes $i$ and $j$, then $h(d') = \mathsf{t}$. We easily find that $S'[k] = S[k]$ for $0 \leq k \leq n$ and $k \neq j$ and $S'[j] = 1$. Hence $S'[i] = 0$. Therefore $h(d') = \mathsf{t}$.

3. If the implementation can do the *leader* action, then so can the specification:

   $$(\exists_{i \leq n}\ (S[i] = 0 \wedge \mathit{isempty}(P[i]))) \rightarrow \exists_{i \leq n}\ b$$

   Trivial.

4. If the specification can do the *leader* action and the implementation cannot do an internal action, then the implementation must be able to do the *leader* action:

   $$(\exists_{i \leq n}\ b) \wedge FC \rightarrow (\exists_{i \leq n}\ (S[i] = 0 \wedge \mathit{isempty}(P[i])))$$

   The specification can do the *leader* action if it is in a state where its only parameter $b$ equals $\mathsf{t}$. This means that for the corresponding state $d$ in the implementation we have $S[i] \neq 1$ for some $i \leq n$. Using invariant $\mathrm{I}_1$ we thus obtain $S[i] = 0$. Now we only have to show that $\mathit{isempty}(P[i])$.

   So suppose that $\neg \mathit{isempty}(P[i])$. Let $i_1 \in P[i]$. From invariant $\mathrm{I}_5$ it follows that $S[i_1] = 0 \wedge i \in P[i_1]$. From $FC \wedge S[i] = 0 \wedge S[i_1] = 0 \wedge i_1 \in P[i]$ it follows that $P[i_1] \neq \{i\}$. Thus there exists $i_2 \in P[i_1]$, $i_2 \neq i$ such that $S[i_2] = 0 \wedge i_1 \in P[i_2]$. In this way an infinite sequence $i = i_0, i_1, i_2, i_3, \ldots$ can be constructed such that for all $k$:*Nat* we have $S[k] = 0 \wedge i_k \in P[i_{k+1}] \wedge i_k \neq i_{k+2}$. By $\mathrm{I}_2$ we see that this infinite path is also a path in $P_0$. This contradicts *GoodTopology*.

5. The implementation and the specification perform external actions with the same parameter. Trivial; the action *leader* involves no data.

6. After the implementation and the specification perform the *leader* action, the mapping $h$ still holds: if the implementation can reach data state $d'$ by the execution of the *leader* action, then $h(d') = \mathsf{f}$.

   Assume that the implementation can perform the *leader* action: i.e. $S[i] = 0 \wedge \mathit{isempty}(P[i])$ for some $i \leq n$. Then also the specification can do the *leader* action by item 3. Hence $b = \mathsf{t}$. After the execution of the leader action the state of the specification is given by $b = \mathsf{f}$. Then by Lemma 6.6 we see that all nodes other than $i$ are in state 1. We also see that by the execution of the *leader* action the state of node $i$ becomes 1. So after the action all nodes are in state 1, so then the value of $h$ will be $\mathsf{f}$.

By Theorem 3.11 it follows that for all $n, P, S$

$$\begin{aligned} I(n,P,S) \;\rightarrow\; & \mathit{L\text{-}Imp}(n,P,S) \lhd \mathit{FC}(n,P,S) \rhd \tau \cdot \mathit{L\text{-}Imp}(n,P,S) \\ & = \mathit{L\text{-}Spec}(h(n,P,S)) \lhd \mathit{FC}(n,P,S) \rhd \tau \cdot \mathit{L\text{-}Spec}(h(n,P,S)). \end{aligned}$$

Instantiation of this equation gives

$$\begin{aligned} I(n,P_0,S_0) \;\rightarrow\; & \mathit{L\text{-}Imp}(n,P_0,S_0) \lhd \mathit{FC}(n,P_0,S_0) \rhd \tau \cdot \mathit{L\text{-}Imp}(n,P_0,S_0) \\ & = \mathit{L\text{-}Spec}(h(n,P_0,S_0)) \lhd \mathit{FC}(n,P_0,S_0) \rhd \tau \cdot \mathit{L\text{-}Spec}(h(n,P_0,S_0)), \end{aligned}$$

which reduces to

$$\mathit{L\text{-}Imp}(n,P_0,S_0) \lhd n = 0 \rhd \tau \cdot \mathit{L\text{-}Imp}(n,P_0,S_0) = \mathit{L\text{-}Spec}(\mathtt{t}) \lhd n = 0 \rhd \tau \cdot \mathit{L\text{-}Spec}(\mathtt{t}).$$

# 7 Confluence for process verification

## 7.1 Introduction

In his seminal book [Mil80] Milner devotes a chapter to the notions strong and observation confluence in process theory. Many other authors have confirmed the importance of confluence. E.g. in [HP94, Qin91] the notion is used for on-the-fly reduction of finite state spaces and in [Mil80] it has been used for the verification of protocols.

We felt that a more general treatment of the notion of confluence is in order. The first reason for this is that the treatment of confluence has always been somewhat ad hoc in the setting of process theory. This strongly contrasts with for instance term rewriting, where confluence is one of the major topics. In particular, we want to clarify the relation with *$\tau$-inertness*, which says that if $s \xrightarrow{\tau} s'$, then $s$ and $s'$ are branching bisimilar.

The second reason is that we want to develop systematic ways to prove distributed systems correct in a precise and formal fashion. In this way we want to provide techniques to construct fault free distributed systems. For this purpose the language $\mu$CRL is used. Experience with several protocols gave rise to the development of new and the adaptation of existing techniques to make systematic verification possible [BG94a, BG94b]. Employing confluence also belongs to these techniques. It appears to enable easier verification of distributed systems, which in essence boils down to the application of $\tau$-inertness.

In Section 7.2, we address the relationship between confluence and $\tau$-inertness on transition systems. We introduce strong and weak confluence. We establish that strong confluence implies $\tau$-inertness and we establish that, for convergent transition systems, weak confluence implies $\tau$-inertness.

To be able to deal with systems with idle loops, for instance communication protocols over unreliable channels, we distinguish between progressive and non progressive internal actions. This leads to a notion of weak progressive confluence that only considers the progressing internal actions. We find that weak progressive confluence is enough to guarantee $\tau$-inertness for transition systems that are convergent with respect to progressing internal steps.

In Section 7.3, we direct our attention to establishing confluence. It does hardly make sense to establish confluence directly on transition systems, because these are generally far too large to be represented. Therefore, we try to establish confluence on processes described by LPEs [BG94b] because these can represent large transition systems in a compact symbolic way. In Section 7.4, we show how we can use $\tau$-inertness and confluence to reduce state spaces both on transition systems and on linear processes. Finally, we provide an example illustrating that the application of confluence often reduces the size of state spaces considerably and simplifies the structure of distributed systems, while preserving branching bisimulation. This is in general a very profitable preprocessing step before analysis, testing or simulation of a distributed system.

## 7.2 Confluence and $\tau$-inertness

Throughout this section we fix the set of actions $\mathsf{A}$, which contains an internal action $\tau$.

**Definition 7.1** A *transition system* is a pair $(S, \longrightarrow\!\triangleright)$ with $S$ a set and $\longrightarrow\!\triangleright \subseteq S \times \mathsf{A} \times S$. The set of triples $\longrightarrow\!\triangleright$ induces a binary relation $\xrightarrow{a}\!\triangleright \subseteq S \times S$ for each $a \in \mathsf{A}$ as follows: for all $s, t \in S$ we have $(s,t) \in \xrightarrow{a}\!\triangleright$ iff $(s,a,t) \in \longrightarrow\!\triangleright$. We write $s \xrightarrow{a}\!\triangleright t$ instead of $(s,a,t) \in \longrightarrow\!\triangleright$ and $(s,t) \in \xrightarrow{a}\!\triangleright$. The relation $\xrightarrow{a^*}\!\triangleright \subseteq S \times S$ denotes the reflexive, transitive closure of the relation $\xrightarrow{a}\!\triangleright$.

A transition system $(S, \longrightarrow\!\triangleright)$ is called *convergent* iff there is no infinite sequence of the form $s_1 \xrightarrow{\tau}\!\triangleright s_2 \xrightarrow{\tau}\!\triangleright s_3 \xrightarrow{\tau}\!\triangleright \cdots$.

A relation $R \subseteq S \times S'$ is called a *branching bisimulation* on $(S, \longrightarrow\!\triangleright)$ and $(S', \longrightarrow\!\blacktriangleright)$ iff for all $s \in S$ and $s' \in S'$ such that $sRs'$ we have

1. $s \xrightarrow{a}\!\triangleright t \rightarrow (a = \tau \wedge tRs') \vee (\exists_{u,u'}\ (s' \xrightarrow{\tau^*}\!\blacktriangleright u \xrightarrow{a}\!\blacktriangleright u' \wedge sRu \wedge tRu'))$, and
2. $s' \xrightarrow{a}\!\blacktriangleright t' \rightarrow (a = \tau \wedge sRt') \vee (\exists_{u,u'}\ (s \xrightarrow{\tau^*}\!\triangleright u \xrightarrow{a}\!\triangleright u' \wedge uRs' \wedge u'Rt'))$.

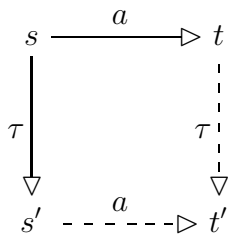
We say that $R$ is a branching bisimulation on $(S, \longrightarrow\!\triangleright)$ iff $R$ is a branching bisimulation on $(S, \longrightarrow\!\triangleright)$ and $(S, \longrightarrow\!\triangleright)$. The union of all branching bisimulations is denoted as $\leftrightarrows_b$.

Next, we present three different notions of confluence on transition systems, namely *strong confluence*, *weak confluence* and *weak progressive confluence*. We investigate whether or not these different notions of confluence are strong enough to serve as a condition for

$$s \xrightarrow{\tau} t \rightarrow s \leftrightarrows_b t \tag{6}$$

to hold. Transition systems that satisfy equation (6) for all $s, t \in S$ are called *$\tau$-inert with respect to $\leftrightarrows_b$*.

**Definition 7.2** A transition system $(S, \longrightarrow)$ is called *strongly confluent* iff for all pairs $s \xrightarrow{a} t$ and $s \xrightarrow{\tau} s'$ of different steps there exists a state $t'$ such that $t \xrightarrow{\tau} t'$ and $s' \xrightarrow{a} t'$. In a diagram:




Omitting the word 'different' in Definition 7.2 would give a stronger notion. This can be seen as follows: the transition system represented by $s \xrightarrow{\tau} t$ is strongly confluent, but would not be strongly confluent when the word 'different' was omitted.

**Theorem 7.3** *Strongly confluent transition systems are $\tau$-inert with respect to $\leftrightarrows_b$.*

The converse of Theorem 7.3 is obviously not valid. A transition system that is $\tau$-inert with respect to $\leftrightarrows_b$, is not necessarily strongly confluent. As a counter-example one can take the transition system

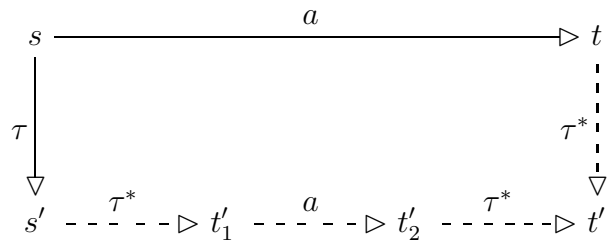
$$t \xleftarrow{\tau} s \xrightarrow{\tau} u. \tag{7}$$

This counter-example means that strong confluence is actually a stronger notion than we need since we are primarily interested in $\tau$-inertness (with respect to $\leftrightarrows_b$). Hence we introduce a weaker notion of confluence, which differs from strong confluence in that it allows zero or more $\tau$-steps in the paths from $t$ to $t'$ and from $s'$ to $t'$.

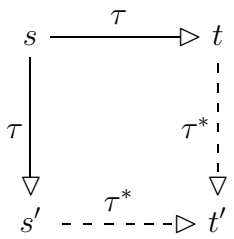
**Definition 7.4** A transition system $(S, \longrightarrow)$ is called *weakly confluent* iff for each pair $s \xrightarrow{a} t$ and $s \xrightarrow{\tau} s'$ of different steps one of the following holds:

- there exist states $t', t'_1, t'_2$ such that $t \xrightarrow{\tau^*} t'$ and $s' \xrightarrow{\tau^*} t'_1 \xrightarrow{a} t'_2 \xrightarrow{\tau^*} t'$. In a

diagram:




- $a = \tau$ and there exists a state $t'$ such that $t \xrightarrow{\tau^*} t'$ and $s' \xrightarrow{\tau^*} t'$. In a diagram:




Weak confluence is too weak to serve as a condition for equation (6) to hold, i.e. weakly confluent transition systems are not necessarily $\tau$-inert with respect to $\leftrightarrows_b$. However, if we also assume that the transition system is convergent, then weak confluence implies $\tau$-inertness.

**Theorem 7.5** *Let $(S, \longrightarrow\!\!\triangleright)$ be convergent and weakly confluent, then $(S, \longrightarrow\!\!\triangleright)$ is $\tau$-inert with respect to $\leftrightarrows_b$.*

Theorem 7.5 relies on convergence of the transition system in question. However, many realistic examples of protocol specifications correspond to transition systems that are not convergent. As soon as a protocol internally consists in some kind of correction mechanism (e.g. retransmissions in a data link protocol) the specification of that protocol will contain an idle loop.

Since we strongly believe in the importance of applicability to realistic examples, we considered the requirement that the transition system is convergent a serious drawback. Therefore, we distinguish, as in Section 3.4, between progressive internal actions, denoted by $\tau_>$ and non-progressive internal actions, denoted by $\tau_<$. This enables us to formulate a notion of confluence, which is sufficiently strong for our purposes and only relies on convergence of the progressive $\tau$-steps.

**Convention 7.6** We use the following notations:

- $s \xrightarrow{\tau_>} t$ for a progressive $\tau$-step from $s$ to $t$,
- $s \xrightarrow{\tau_<} t$ for a non-progressive $\tau$-step from $s$ to $t$,

- $s \xrightarrow{\tau} \triangleright\ t$ for $s \xrightarrow{\tau_>} \triangleright\ t$ or $s \xrightarrow{\tau_<} \triangleright\ t$.

From now on we try to prove $\tau_>$*-inertness with respect to* $\leftrightarrows_b$, instead of $\tau$-inertness with respect to $\leftrightarrows_b$. In a formula:

$$s \xrightarrow{\tau_>} \triangleright\ t \rightarrow s \leftrightarrows_b t \tag{8}$$

It should be noted that the definition of branching bisimulation is not affected by the distinction of progressive and non-progressive $\tau$-steps. We first provide the definition of weak $>$-confluence.

**Definition 7.7** A transition system $(S, \longrightarrow\triangleright)$ is called *weakly* $>$*-confluent* (pronounce: weakly progressive confluent) iff for each pair $s \xrightarrow{\tau_>} \triangleright\ s'$ and $s \xrightarrow{a} \triangleright\ t$ of different steps one of the following holds:

- there exist states $t', s'_1, s'_2$ such that $t \xrightarrow{\tau^*_>} \triangleright\ t'$ and $s \xrightarrow{\tau^*_>} \triangleright\ s'_1 \xrightarrow{a} \triangleright\ s'_2 \xrightarrow{\tau^*_>} \triangleright\ t'$, or
- $a = \tau$ and there exists a state $t$ such that $t \xrightarrow{\tau^*_>} \triangleright\ t'$ and $s \xrightarrow{\tau^*_>} \triangleright\ t$.

**Theorem 7.8** *Let* $(S, \longrightarrow\triangleright)$ *be* $>$*-convergent and weakly* $>$*-confluent, then* $(S, \longrightarrow\triangleright)$ *is* $\tau_>$*-inert with respect to* $\leftrightarrows_b$.

## 7.3 Confluence of Linear Process Equations

We want to use the notion confluence to verify the correctness of processes. In order to do so, we must be able to determine whether a transition system is confluent. This is in general not possible, because the transition systems belonging to distributed systems are often too large to be handled as plain objects. In order to manipulate with large state spaces, processes described by C-LPEs can be used as in these the state space is compactly encoded using data parameters.

In this section we describe how a C-LPE can be shown to be confluent. In the next section we show how confluence is used to reduce the size of state spaces.

Recall the definition of a clustered linear process equation of Definition 3.7. A clustered linear process equation is an expression of the form

$$p(d) = \sum_{a \in Act} \sum_{e_a : E_a} a(f_a(d, e_a)) \cdot p(g_a(d, e_a)) \lhd b_a(d, e_a) \rhd \delta \tag{9}$$

We assume that the internal action $\tau$ ($\tau_>$ and $\tau_<$ if progressing and non-progressing $\tau$'s are distinguished) has no data parameter.

It is straightforward to see how a linear process equation determines a transition system. The process $p(d)$ can perform an action $a(f_a(d, e_a))$ for every $a \in Act$ and every data

element $e_a$ of sort $E_a$, provided the condition $b_a(d, e_a)$ holds. The process then continues as $p(g_a(d, e_a))$. Hence, the notions defined in the previous sections carry over directly. Thus, if $b_a(d, e_a)$ holds then

$$p(d) \xrightarrow{a(f_a(d,e_a))} p(g_a(d, e_a)).$$

As we distinguish between progressing and non-progressing $\tau$'s, we use the notion convergence with respect to the progressing $\tau$'s (i.e. $\tau_>$).

**Definition 7.9** A clustered linear process equation as given in equation (9) is called $>$-*convergent* iff there is a well-founded ordering $<$ on $D$ such that for all $a \in Act$ with $a = \tau_>$ and for all $d{:}D$, $e_a{:}D_a$ we have that $b_a(d, e_a)$ implies $g_a(d, e_a) < d$.

Note that this definition of progressive convergence is in line with the normal notion of convergence (Definition 3.3) in the sense that only the progressive $\tau$-steps are considered.

We provide sufficient criteria for $p$ to be strongly confluent. Let $p$ be the clustered linear process equation given in equation (9).

**Theorem 7.10** *The process $p$ as defined in equation (9) is strongly confluent if for all $d{:}D$, $a \in Act$, $e_a{:}E_a$, $e_\tau{:}E_\tau$ such that*

(i) *if $a = \tau$ then $g_a(d, e_a) \neq g_\tau(d, e_\tau)$,*

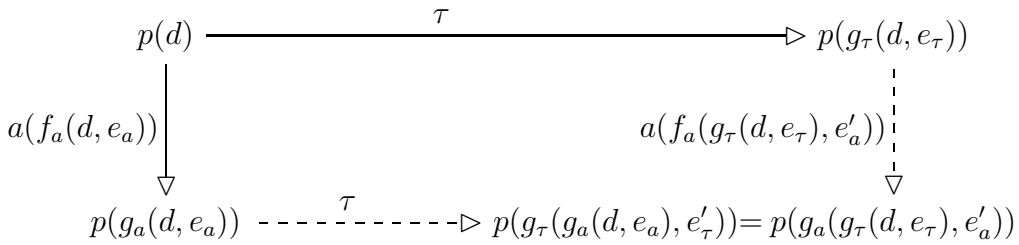
(ii) *$b_a(d, e_a)$, and*

(iii) *$b_\tau(d, e_\tau)$*

*the following property holds: there exist $e'_a{:}E_a$ and $e'_\tau{:}E_\tau$ such that*

$$\begin{array}{ll} & f_a(d, e_a) = f_a(g_\tau(d, e_\tau), e'_a) \\ \wedge & b_a(g_\tau(d, e_\tau), e'_a) \\ \wedge & b_\tau(g_a(d, e_a), e'_\tau) \\ \wedge & g_a(g_\tau(d, e_\tau), e'_a) = g_\tau(g_a(d, e_a), e'_\tau). \end{array}$$

The criteria can best be understood via the following diagram.



Note that in this diagram $p(g_a(d,e_a))$ and $p(g_\tau(d,e_\tau))$ are supposed to be different if $a=\tau$ (see condition (i) in the above definition).

Now that we have derived a rather simple condition for strong confluence we turn our attention to weak progressive confluence. This is more involved, because we must now speak about sequences of transitions. In order to keep notation compact, we introduce some notation. Let $\sigma,\sigma',\ldots$ range over lists of pairs $\langle a,e_a\rangle$ with $a\in Act$ and $e_a{:}E_a$. We define $\mathcal{G}_d(\sigma)$ with $d\in D$ by induction over the length of $\sigma$:

$$\begin{array}{lcl}\mathcal{G}_d(\lambda) & = & d,\\ \mathcal{G}_d(\sigma\langle a,e_a\rangle) & = & g_a(\mathcal{G}_d(\sigma),e_a).\end{array}$$

Each $\sigma$ determines an execution fragment:

$$\underbrace{p(d)\longrightarrow\!\!\triangleright p(\mathcal{G}_d(\sigma))}_{\text{determined by }\sigma}\xrightarrow{a(f_a(\mathcal{G}_d(\sigma),e_a))}\!\!\triangleright\ p(\mathcal{G}_d(\sigma\langle a,e_a\rangle))$$

is the execution fragment determined by $\sigma\langle a,e_a\rangle$. This execution fragment is allowed for $p(d)$ iff the conjunction $\mathcal{B}_d(\sigma)$ of all conditions associated to the actions in $\sigma$ evaluates to *true*. The boolean $\mathcal{B}_d(\sigma)$ is also defined by induction to the length of $\sigma$:

$$\begin{array}{lcl}\mathcal{B}_d(\lambda) & = & true,\\ \mathcal{B}_d(\sigma\langle a,e_a\rangle) & = & \mathcal{B}_d(\sigma)\wedge b_a(\mathcal{G}_d(\sigma),e_a).\end{array}$$

We write $\pi_1(\sigma)$ for the sequence of actions that is obtained from $\sigma$ by applying the first projection to all its elements. E.g. $\pi_1(\langle a,e_a\rangle\langle b,e_b\rangle)=ab$.

In the following theorem we provide sufficient criteria for a C-LPE $p$ to be weakly $>$-confluent. Due to its generality the theorem looks rather complex. However, in those applications that we considered, the lists that are existentially quantified were mainly empty, which trivialises the main parts of the theorem.

**Theorem 7.11** *The process $p$ as defined in equation (9) is weakly $>$-confluent if $p$ is $>$-convergent and for all $d{:}D$, $a\in Act$, $e_a{:}E_a$, $e_{\tau_>}{:}E_{\tau_>}$ such that*

(i) *if $a=\tau_>$ then $g_a(d,e_a)\neq g_{\tau_>}(d,e_\tau)$,*

(ii) *$b_a(d,e_a)$, and*

(iii) *$b_{\tau_>}(d,e_{\tau_>})$*

*the following property holds: there exist $\sigma_1,\sigma_2,\sigma_3$ and $e'_a{:}E'_a$ such that*

$$\begin{array}{ll} & \pi_1(\sigma_i)=\tau_>^*\ \textit{for all } i=1,2,3\\ \wedge & f_a(d,e_a)=f_a(\mathcal{G}_{g_{\tau_>}(d,e_{\tau_>})}(\sigma_1),e'_a)\\ \wedge & \mathcal{B}_{g_a(d,e_a)}(\sigma_3)\\ \wedge & \mathcal{B}_{g_{\tau_>}(d,e_{\tau_>})}(\sigma)\\ \wedge & \mathcal{G}_{g_a(d,e_a)}(\sigma_3)=\mathcal{G}_{g_{\tau_>}(d,e_{\tau_>})}(\sigma),\end{array}$$

*where* $\sigma = \sigma_1\langle a, e'_a\rangle\sigma_2$, *or* $a = \tau$ *and* $\sigma = \sigma_1\sigma_2$.

## 7.4 State Space Reduction

In this section, we employ the results about confluence and $\tau$-inertness that we have obtained thus far to achieve state space reductions and to simplify the behaviour of processes. First, we present the results on transition systems in general, and then on linear process equations. This is done as for transition systems the results are easy to understand. However, as argued in the previous section, the results can be applied more conveniently in the setting of linear process equations.

**Definition 7.12** Let $T_1 = (S, \longrightarrow\!\!\triangleright)$ and $T_2 = (S, \longrightarrow\!\!\blacktriangleright)$ be transition systems. We call $T_2$ a *Tau-Prioretised-reduction* (*TP-reduction*) of $T_1$ iff

(i) $\longrightarrow\!\!\blacktriangleright \subseteq \longrightarrow\!\!\triangleright$,

(ii) for all $s, s' \in S$ if $s \xrightarrow{a}\!\!\triangleright\ s'$ then $s \xrightarrow{a}\!\!\blacktriangleright\ s'$ or $s \xrightarrow{\tau_>}\!\!\blacktriangleright\ s''$ for some $s''$.

Clearly, $T_2$ can be obtained from $T_1$ by iteratively removing transitions from states as long as these keep at least one outgoing progressive $\tau$-step. It does not need any comment that this may considerably reduce the state space of $T_1$, especially because large parts may become unreachable.

The following theorem says that if $T_1$ is $\tau_>$-inert with respect to $\leftrightarrows_b$, then a TP-reduction maintains branching bisimulation. As confluence implies $\tau$-inertness, this theorem explains how confluence can be used to reduce the size of transition systems.

**Theorem 7.13** *Let* $T_1 = (S, \longrightarrow\!\!\triangleright)$ *and* $T_2 = (S, \longrightarrow\!\!\blacktriangleright)$ *be transition systems. If* $T_1$ *is* $\tau_>$*-inert with respect to* $\leftrightarrows_b$ *and* $T_2$ *is a* $>$*-convergent TP-reduction of* $T_1$ *then for each state* $s \in S$: $s \leftrightarrows_b s$.

As has been shown in the previous section, weak $>$-confluence can relatively easy be determined on C-LPEs. We provide a way to reduce the complexity of a C-LPE. Below we reformulate the notion of a TP-reduction on C-LPEs. We assume that $p$ is a C-LPE according to equation (9) and that the data sort $E_\tau$ is ordered by some total ordering $\prec$.

**Definition 7.14** The *TP-reduction* of $p$ is the linear process

$$p_r(d) = \sum_{a \in Act} \sum_{e_a : E_a} a(f_a(d, e_a)) \cdot p_r(g_a(d, e_a)) \lhd b_a(d, e_a) \wedge c_a(d, e_a) \rhd \delta$$

where

$$c_a(d, e_a) = \begin{cases} \neg\exists_{e_{\tau_>} : E_{\tau_>}}\ b(d, e_{\tau_>}) & \text{if } a \neq \tau_>, \\ \neg\exists_{e_{\tau_>} : E_{\tau_>}}\ (e_a \prec e_{\tau_>} \wedge b(d, e_{\tau_>})) & \text{if } a = \tau_>. \end{cases}$$

Note that for the sake of conciseness, we use an existential quantification in the condition $c_a(d, e_a)$, which does not adhere to the formal definition of $\mu$CRL.

**Theorem 7.15** *If the linear process $p$ is $>$-convergent and weakly $>$-confluent, then for all $d{:}D$*

$$p(d) \leftrightarrows_b p_r(d).$$

## 7.5 An Example: Concatenation of Two Queues

We illustrate how we apply the theory by means of an example, where the structure of the processes is considerably simplified by a confluence argument. Consider the following linear process $Q(q_1, q_2)$ describing the concatenation of two queues $q_1$ and $q_2$. The architecture of this process is given in Figure 3.

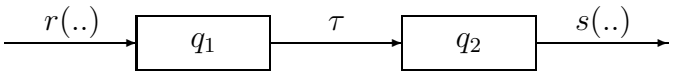


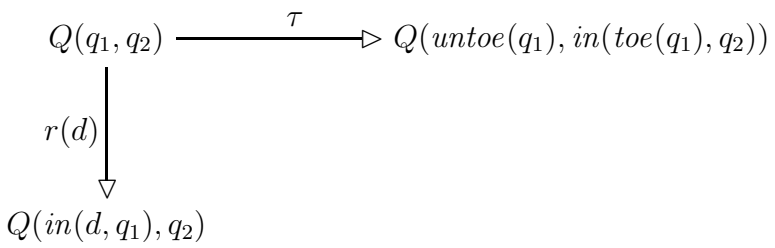

Figure 3: Architecture of the concatenation of the queues.

$$\begin{array}{rcl} Q(q_1, q_2) & = & \sum_{d:D} r(d) \cdot Q(in(d, q_1), q_2) \lhd true \rhd \delta \\ & + & \tau \cdot Q(untoe(q_1), in(toe(q_1), q_2)) \lhd \neg isempty(q_1) \rhd \delta \\ & + & s(toe(q_2)) \cdot Q(q_1, untoe(q_2)) \lhd \neg isempty(q_2) \rhd \delta \end{array}$$

As we can see, the process $Q(q_1, q_2)$ can always read a datum and insert it in $q_1$. If $q_2$ is not empty then the 'toe' of $q_2$ can be sent. The internal action $\tau$ removes the first element of $q_1$ and inserts it in $q_2$.

Using Theorem 7.10 we can straightforwardly prove that $Q(q_1, q_2)$ is strongly confluent.

Let us consider the strong confluence in more detail with respect to the read action $r(d)$.



This situation can only occur if both the read action $r(d)$ and the $\tau$ action are enabled: `t` and $\neg isempty(q_1)$. To establish strong confluence in this specific case we must find:

$$\begin{array}{ccc} & & Q(untoe(q_1), in(toe(q_1), q_2)) \\ & & \vdots\, r(d') \\ & & \downarrow \\ & & Q(in(d', untoe(q_1)), in(toe(q_1), q_2)) \\ & & = \\ Q(in(d, q_1), q_2) & \overset{\tau}{\dashrightarrow} & Q(untoe(in(d, q_1)), in(toe(in(d, q_1)), q_2)) \end{array}$$

for some $d'{:}D$ such that the data parameters of the two read actions are equal ($d = d'$). Furthermore, we need that the states resulting after the execution of these actions are equal. The state after the $r(d')$ action is given by $(in(d', untoe(q_1)), in(toe(q_1), q_2))$ and after the $\tau$ action by $(untoe(in(d, q_1)), in(toe(in(d, q_1)), q_2))$ respectively. With the observation that the equality of states is defined to be pairwise equality we obtain the following condition: for all queues $q_1, q_2$ and $d{:}D$

$$\begin{array}{lll} \neg isempty(q_1) \rightarrow \exists_{d':D} & ( & d = d' \\ & \wedge & \neg isempty(in(d, q_1)) \\ & \wedge & in(d', untoe(q_1)) = untoe(in(d, q_1)) \\ & \wedge & in(toe(q_1), q_2) = in(toe(in(d, q_1)), q_2) \\ & ). & \end{array}$$

Similarly, we can formulate the following conditions for the action $s$. For all queues $q_1, q_2$

$$\begin{array}{lll} \neg isempty(q_2) \wedge \neg isempty(q_1) \rightarrow & ( & toe(q_2) = toe(in(toe(q_1), q_2)) \\ & \wedge & \neg isempty(in(toe(q_1), q_2)) \\ & \wedge & \neg isempty(q_1) \\ & \wedge & untoe(q_1) = untoe(q_1) \\ & \wedge & untoe(in(toe(q_1), q_2)) = in(toe(q_1), untoe(q_2)) \\ & ). & \end{array}$$

With the appropriate axioms for queues, the validity of these facts is easily verified.

For the $a = \tau$ we find that the precondition $a = \tau \rightarrow g_a(d, e_a) \neq g_\tau(d, e_\tau)$ is instantiated to $\tau = \tau \rightarrow (untoe(q_1) \neq untoe(q_1) \vee in(toe(q_1), q_2) \neq in(toe(q_1), q_2))$, which is a contradiction.

Now, by Theorem 7.15, the following TP-reduced version (see Definition 7.14) of $Q(q_1, q_2)$

is branching bisimilar to $Q(q_1, q_2)$.

$$\begin{array}{rcl} Q_r(q_1,q_2) & = & \sum_{d:D} r(d) \cdot Q_r(in(d,q_1),q_2) \lhd isempty(q_1) \rhd \delta \\ & + & \tau \cdot Q_r(untoe(q_1), in(toe(q_1),q_2)) \lhd \neg isempty(q_1) \rhd \delta \\ & + & s(toe(q_2)) \cdot Q_r(q_1, untoe(q_2)) \lhd \neg isempty(q_2) \wedge isempty(q_1) \rhd \delta \end{array}$$

Note that after the TP-reduction $q_1$ never contains more than one element!

# References


[Bar81] H.P. Barendregt. *The lambda calculus*. North-Holland, 1981.

[BBG97] M.A. Bezem, R.N. Bol, and J.F. Groote. Formalizing process algebraic verifications in the calculus of constructions. *Formal Aspects of Computing*, 9:1–48, 1997.

[Bek71] H. Bekič. Towards a mathematical theory of processes. Technical Report TR25.125, IBM Laboratory, Vienna, 1971.

[BG94a] M.A. Bezem and J.F. Groote. A correctness proof of a one-bit sliding window protocol in $\mu$CRL. *The Computer Journal*, 37(4):289–307, 1994.

[BG94b] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *CONCUR'94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 401–416, Uppsala, Sweden, 1994. Springer-Verlag.

[BHP97] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. Grid protocols based on synchronous communication. *Science of Computer Programming*, 29:199–233, 1997.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information & Control*, 60(1-3):109–137, 1984.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[CM89] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.

[CWI00] CWI. *$\mu$CRL: a language and toolset to study communicating processes with data*, 2000. `http://www.cwi.nl/~mcrl/`.

[dBdV96] J.W. de Bakker and E. de Vink. *Control Flow Semantics*. MIT Press, 1996.

[DFH+93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. werner. The Coq proof assistant user's guide, version 5.9. Technical report, INRIA-Rocquencourt and CNRS-ENS Lyon, 1993.

[DGRV97] M.C.A. Devillers, W.O.D. Griffioen, J.M.T. Romijn, and F.W. Vaandrager. Verification of a leader election protocol – formal methods applied to IEEE 1394. Technical report, Computing Science Institute, University of Nijmegen, 1997.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[FGK+96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CAESAR?ALDEBARAN Development Package): A protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings of CAV'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer-Verlag, 1996.

[FGK97] L.-Å. Fredlund, J.F. Groote, and H. Korver. Formal verification of a leader election protocol in process algebra. *Theoretical Computer Science*, 177:459–486, 1997.

[GK94] J.F. Groote and H. Korver. Correctness proof of the bakery protocol in $\mu$CRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 63–86. Springer-Verlag, 1994.

[GMS97] J.F. Groote, F. Monin, and J. Springintveld. A computer checked algebraic verification of a distributed summing protocol. Technical Report 97/14, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1997.

[GMvdP98] J.F. Groote, F. Monin, and J. van de Pol. Checking verifications of protocols and distributed systems by computer. In D. Sangiorgi and R. de Simone, editors, *Proceedings CONCUR'98*, volume 1466 of *Lecture Notes in Computer Science*, pages 629–655. Sophia Antipolis, Springer-Verlag, 1998.

[GP94] J.F. Groote and A. Ponse. Proof theory for $\mu$CRL: A language for processes with data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages, Proceedings of the International Workshop on Semantics of Specification Languages, Utrecht, The Netherlands, 25-27 October 1993*, Workshops in Computing, pages 232–251. Springer-Verlag, 1994.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of $\mu$CRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing, pages 26–62. Springer-Verlag, 1995.

[Gro97a] J.F. Groote. A note on $n$ similar parallel processes. In S. Gnesi and D. Latella, editors, *ERCIM Workshop on Formal Methods for Industrial Critical Systems*, pages 65–75. Cesena, Italy, 1997.

[Gro97b] J.F. Groote. The syntax and semantics of timed $\mu$CRL. Technical Report SEN-R9709, CWI, Amsterdam, 1997.

[GS95] J.F. Groote and J. Springintveld. Focus points and convergent process operators. A proof strategy for protocol verification. Technical Report 142, University Utrecht, Department of Philosophy, 1995.

[GS96] J.F. Groote and M.P.A. Sellink. Confluence for process verification. *Theoretical Computer Science*, 170(1-2):47–81, 1996.

[GvdP96] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. A case study in computer checked verification. In M. Wirsing and M. Nivat, editors, *Proceedings of AMAST'96*, volume 1101 of *Lecture Notes in Computer Science*, pages 536–550, Munich, Germany, 1996.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall International, 1985.

[HP94] G.J. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings FORTE'94*, Berne, Switzerland, 1994.

[IEE96] IEEE Computer Society. *IEEE Standard for a high performance serial bus*, 1996.

[IT94] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*. ITU-T, Geneva, June 1994.

[Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.

[KRR98] J.J.T. Kleijn, M.A. Reniers, and J.E. Rooda. A process algebra based verification of a production system. In J. Staples, M.G. Hinchley, and S. Liu, editors, *Second IEEE Conference on Formal Engineering Methods*, pages 90–99, Brisbane, Australia, December 1998. IEEE Computer Society Press.

[KS94] H. Korver and J. Springintveld. A computer-checked verification of Milner's scheduler. In M. Hagiya and J.C. Mitchell, editors, *Proceedings of the international symposium on Theoretical Aspects of Computer Software (TACS'94)*, volume 789 of *Lecture Notes in Computer Science*, pages 161–178, Sendai, Japan, 1994. Springer-Verlag.

[Lut97] S.P. Luttik. Description and formal specification of the link layer of P1394. Technical Report SEN-R9706, CWI, Amsterdam, 1997.

[Lyn96] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Maz88] A. Mazurkiewicz. Basic notions of trace theory. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear time, branching time and partial orders in logics and models for concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 285–363. Springer-Verlag, 1988.

[Mil73] R. Milner. A mathematical model of computing agents. In H.E. Rose and J.C. Shepherdson, editors, *Proceedings Logic Colloquium 1973*, pages 158–173. North-Holland, 1973.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall International, 1989.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[Pau90] L.C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[Phi87] I.C.C. Philips. Refusal testing. *Theoretical Computer Science*, 50:241–284, 1987.

[Qin91] H. Qin. Efficient verification of determinate processes. In J.C.M. Baeten and J.F. Groote, editors, *CONCUR'91, 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 471–494, Amsterdam, The Netherlands, 1991. Springer-Verlag.

[Rei85] W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs in Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[SM97] M. Sighireanu and R. Mateescu. Validation of the link layer protocol of the IEEE-1394 serial bus (FireWire): an experiment with E-LOTOS. Technical Report 3172, INRIA, 1997.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1993.

[SvdZ98] C. Shankland and M.B. van der Zwaag. The tree identify protocol of IEEE 1394 in $\mu$CRL. *Formal Aspects of Computing*, 10:509–531, 1998.

[vG90] R.J. van Glabbeek. The linear time - branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90 - Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, 1990. Springer-Verlag.

[vG93] R.J. van Glabbeek. The linear time - branching time spectrum II. The semantics of sequential processes with silent moves. In E. Best, editor, *CONCUR'93, International Conference on Concurrency Theory*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81, Hildesheim, Germany, 1993. Springer-Verlag.

[vW95] J.J. van Wamel. *Verification techniques for elementary data types and retransmission protocols*. PhD thesis, University of Amsterdam, 1995.

[Win87] G. Winskel. Event structures. In Brauer, Reissig, and G. Rozenberg, editors, *Petri nets: Applications and relationships to other models of concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer-Verlag, 1987.

# Index