

# THE CONCURRENCY COLUMN

BY

**LUCA ACETO**

BRICS, Department of Computer Science  
Aalborg University, 9220 Aalborg Ø, Denmark  
luca@cs.auc.dk, <http://www.cs.auc.dk/~luca/BEATCS>

This issue of the Concurrency Column is devoted to a piece on the need for proof methodologies in process algebra. This contribution by Wan Fokkink, Jan Friso Groote and Michel Reniers is inspired by one of the most classic applications of process algebra, viz. the verification of the correctness of protocols specified using a process algebraic formalism, and in particular recounts how the authors struggled with a correctness proof for the most complex sliding window protocol presented in Tanenbaum's *Computer Networks* textbook which led to the development of several useful proof techniques. The authors have a deep knowledge of this topic, as witnessed by, e.g., their many publications and a chapter on this topic in the *Handbook of Process Algebra*. I thank them for sharing their knowledge and thoughts on the status of process algebraic approaches to protocol verification with the community via this column. The main message of their contribution is that, after more than a decade of experience, it is more obvious than ever that the mathematics of process algebraic verification has to be developed much further. Let's get down to work!

I take this opportunity for reminding the readers that the submission deadline for CONCUR 2004, the 15th International Conference on Concurrency Theory, is Friday, 9 April 2004. I hope that many of you will submit a paper to that event. CONCUR is, after all, the flagship conference of our community. See <http://www.doc.ic.ac.uk/concur2004/> for updated information on this event and its satellite activities.

Finally, I strongly encourage those of you who are interested in organizing a thematic research workshop, a strategic meeting charting new research agenda or an advanced school on a theme related to concurrency theory to consider the University Residential Centre of Bertinoro, Bertinoro (Forlì), Italy, as a possible location for it. Activities taking place in that beautiful conference

location are held under the organization and sponsorship of BICI (Bertinoro International Center for Informatics). On behalf of the concurrency theory community, I welcome the establishment of such an association devoted to the development of research in Computer Science via the sponsorship of high quality events in an environment that offers excellent support, and a congenial atmosphere, for the hosting of research activities.

## **PROCESS ALGEBRA NEEDS PROOF METHODOLOGY**

Wan Fokkink, Jan Friso Groote, Michel Reniers

Dep. of Mathematics and Computer Science, Eindhoven University of Technology

P.O. Box 513, 5600 MB Eindhoven, The Netherlands

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Email: Wan.Fokkink@cwi.nl, {J.F.Groote,M.A.Reniers}@tue.nl

### **Why is process algebra exciting?**

An early paper by Milner in 1973 [41] gave a clear motivation for process algebra. He gave three reasons to design a process algebra.

- All (computer) systems interact with their environment. For most of these, this is their primary ‘raison d’être’. So, within computer science, we need a formalism in which interaction is a primary citizen.
- Nondeterminism is important. The actual behaviour of a computer system is influenced by factors that we do not understand or are too complex to include in a comprehensive description. For instance the exact moments at which interrupts occur in a computer can have substantial influence on its overall behaviour. Yet, incorporating these moments would quickly make the study of any non-trivial behaviour in the system untenable. Nondeterminism provides a way to manage this complexity. We simply describe nondeterministically that interrupts can occur at any moment. Milner uses the ‘weather’ that can influence the behaviour of computers and that we cannot completely describe and understand.
- Parallelism is an omnipresent notion in computers.

Especially, the first two aspects are missing in many foundational formalisms, in particular in those from the 70’ies of the last century.

Based upon these motivations, Milner developed CCS [42], a simple and very elegant theory for communicating systems. Very similar theories were developed by other researchers among which CSP [34] and ACP [5]. As we grew up in the realm of ACP, we use it in the rest of this note, but all that is being said applies to all process theories in general.

Another new key notion in process algebra is the internal action  $\tau$  (which we also attribute to Milner [42], although we do not know whether this is its true origin). The idea is that the exact nature of most actions that occur in a computer system is irrelevant. Hence, these actions can be made invisible by declaring them hidden. In other words, these actions are renamed to  $\tau$ 's. Using appropriate reduction relations, e.g. weak- [42, 45] or branching bisimulation [21], the residual visible behaviour can be made small and insightful.

Note that besides process algebras, there are more formalisms that allow interaction, nondeterminism and abstraction, such as Petri-nets [46] and I/O-automata [38]. But to our taste, none of these formalisms integrate all these aspects so nicely as process algebra.

## The limits of classical process algebra

The axiomatic theories of CCS, ACP and CSP led to typical and elegant correctness proofs of intricate communicating systems. Examples are for instance Milner's scheduler and the alternating bit protocol (see e.g. [3, 42]).

Unfortunately, these examples turned out to be the limit of what could be proven correct using manipulation by means of the axioms directly. Around 1990, it became obvious that the applicability of process algebra in its original form for practical purposes was limited.

Obviously, if process algebra could not be effectively applied to much more complex systems, there should be only one reasonable destiny for all the work done on it: the shelves of our national libraries. But, given the elegance and simplicity of its elementary theory, we could not accept this fate.

So, in 1990 the challenge was (and for the larger part still is):

Can we develop process algebra such that it can be used effectively to design the correct behaviour of realistic interacting programs and systems?

The first derived question, namely whether we can describe realistic, interacting systems, had already been answered affirmatively by in particular LOTOS [36] and subsequently by other formalisms such as PSF [40]. In these languages the expressiveness of process algebra is enhanced with data types and some syntactic sugaring.

One of the striking conclusions was that specifying interacting systems using these formalisms really improved quality and clarity, compared to the standard verbose and verbal specifications, that unfortunately are still very commonly in use. But after initial enthusiasm, it turned out that especially the larger specifications did contain mistakes similar to the typical mistakes in programs. A specification formalism by itself does not guarantee correctness.

This made it obvious that ways were needed to ascertain the correctness of such specifications.

## **Why are state based techniques not the only answer**

A major movement to provide the means to establish the correctness of specifications originates in state space exploration. The basic idea is that a system specification is transformed into an automaton or state space. By explicit exploration of this state space, properties, such as deadlock freedom, can be established. Using property languages such as the modal  $\mu$ -calculus [10], much more advanced properties can be formulated.

Using spectacular techniques, such as Binary Decision Diagrams [12], partial order reduction [23], bit-hashing [35], supported by the even more spectacular increase of the speed of computers and the use of networks of computers it is now possible to effectively and automatically prove many properties about the behavioural specifications of destitute interacting systems. This is actually leading to modeling as a new paradigm in computer science, which is common in most other engineering disciplines. This means that the essential behaviours of programmed systems are being modeled and that insight is primarily obtained by studying the behaviour of the model.

Unfortunately, many realistic systems and their models exhibit behaviours that lead to state spaces too large and too complex to deal with using any of the methods above. And although we often wanted to believe differently, most of the real world examples to which we want to apply our techniques, are way out of reach. A typical example is a new railway safety control philosophy developed in the Netherlands [4] allowing a more efficient and reliable railway transport. One of the reasons that it will not be taken into service is that nobody can convincingly certify its correct operation. Until this is possible, it seems that the Dutch railway companies will stick to the proven, simple control systems to which they are accustomed, denying customers the benefits of the new technology.

In those cases where automatic methods are insufficiently helpful, manual manipulation can save the day. Typically, protocols and algorithms deal with unspecified or unbounded data, are about an unbounded number of communicating partners and are often quite complex. In order to prove the correctness of such systems

human ingenuity and intuition are indispensable. The ability to include the human intellect into the verification effort will be a distinguishing success factor for many decades to come.

This situation is similar to mathematics, where numerical techniques and automatic formula manipulation play increasingly important roles. But even in engineering mathematics, manual verification techniques, intuition, insight, experience and hard labour remain essential to solve problems. It however needs to be said that the combination of manual and automated techniques have advanced this field as a whole.

Within computer science the situation is similar. So, besides automated techniques, it is essential to further the field of manual verification, far beyond the level of Milner's Scheduler or the alternating bit protocol.

## The sliding window protocol

In 1990 we were ready to undertake this challenge. But first we needed a tractable specification language without any form of syntactic sugar but still sufficiently strong to model complex interacting systems. So, we needed a formalism more advanced than bare process algebra, but not as syntactically rich as LOTOS and PSF. We designed the language  $\mu$ CRL (micro Common Representation Language) [26] consisting of ACP, abstract equational data types and two new operators (the if-then-else and the sum over (infinite) data types) to glue data and processes together.

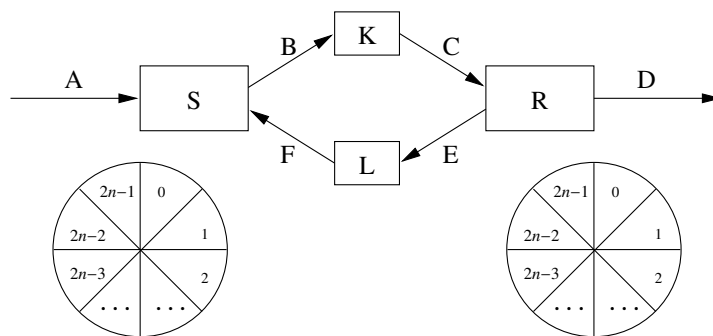


Figure 1: Sliding window protocol

After this, we embarked upon the question whether we could prove the correctness of the most complex sliding window protocol in Tanenbaum's Computer Networks [50, Second edition] solely on the basis of the axioms and rules of process algebra. The structure of the sliding window protocol is drawn in figure 1.

The basic idea behind the protocol is that data must be sent from  $A$  to  $D$ . We only consider the unidirectional variant here. In the bidirectional variant data is sent from  $D$  to  $A$  also. The data is stored in a buffer in  $S$  and is subsequently sent via the channel  $K$  to the receiver  $R$ . The receiver  $R$  sends acknowledgements back to  $S$  via channel  $L$ . While in transit, data and acknowledgements can get lost. Therefore, sender  $S$  regularly resends its data, as long as it has not been acknowledged and receiver  $R$  retransmits acknowledgements periodically. Only if data has been acknowledged, the sender  $S$  removes it from its buffer. A typical feature of the sliding window protocol is that the messages are numbered. By looking at these numbers and by buffering the messages, receiver  $R$  can deliver the data in the same order as they were read by  $S$ . It turns out that the protocol works correctly by numbering the messages modulo  $2n$  if both buffers have size  $n$ .

Algebraically proving the correctness of the sliding window protocol appeared to be much harder than expected, and has only recently been finished [19]. There have been a number of failed attempts to prove the sliding window protocol correct within the process algebraic community. There are some attempts where the nature of the protocol has been substantially adapted [25, 44, 47]. Furthermore, there are a number of proofs of simplified sliding window protocols, with only a single buffer place [7, 52, 53] or where modulo calculation does not essentially play a role [15]. We consider the fact that we managed to give a process algebraic proof of the full sliding window protocol with arbitrary buffer size  $n$  and packet numbering modulo  $2n$  (and had the proof checked using PVS [43]) a substantial step forward. More importantly, we identified several essential techniques that make the proof rather straightforward.

Using assertional techniques a number of successful proofs of the sliding window protocol have been given [13, 48, 49]. Especially, the proof of Anneke Schoone gave us the essential inspiration for a technique which we call ‘coordinate transformations’. Typical for the assertional proofs are the limited correctness properties that can be shown. For the sliding window protocol correctness is formulated as: the list of delivered data of the sliding window protocol matches the list of input data. Within process algebra the more insightful theorem is proven that the behaviour of the sliding window protocol with window size  $n$  is branching bisimilar to that of a queue with buffer size  $2n$ .

Very early on we made two observations about the third sliding window protocol of Tanenbaum. The first was that the external behaviour of the protocol as stated in Tanenbaum is extremely complex. So, we had to make slight adaptations to guarantee that the protocol nicely behaved like a bounded queue. The second observation was that the protocol contained a deadlock, which we had to repair.

In the remainder of this note we first shortly introduce the language  $\mu\text{CRL}$  and subsequently address the techniques that we have identified as important to

algebraically prove the correctness of interacting systems.

## A short primer in $\mu\text{CRL}$

We want to avoid formulas, and we mainly provide main ideas. Readers that want more are referred to for instance [19, 27] or even the draft of a planned book on these techniques [28]. Yet, it might be useful, to briefly sketch the language  $\mu\text{CRL}$ , which has provided the context for our work.

Processes have the following syntax:

$$p ::= a(d_1, \dots, d_n) \mid \tau \mid \delta \mid p + p \mid p \cdot p \mid p \parallel p \mid \tau_I(p) \mid \partial_H(p) \mid \sum_{d:D} p \mid p \triangleleft c \triangleright p.$$

The process  $a(d_1, \dots, d_n)$  is an action, parameterized with data elements. The process  $\tau$  is the internal action, i.e. an action that cannot be directly observed. The process  $\delta$  is inaction or deadlock, i.e. the process that cannot perform any behaviour. The operator  $+$  denotes the choice between two processes, and the operator  $\cdot$  is sequential composition. The sequential composition operator is often omitted. Parallel composition is denoted by  $\parallel$ . Actions can be hidden using the hiding operator  $\tau_I$  where  $I$  is a set of action labels of the actions that are to be renamed to  $\tau$ . The encapsulation operator  $\partial_H$  blocks all actions whose labels are in the set  $H$ . All these operators are standard operators of ACP [3].

The operators that integrate data and processes are the sum operator  $\sum_{d:D} p$  that represents the possibly infinite choice of process  $p$  over all data elements in  $D$ . The process  $p \triangleleft c \triangleright p$  denotes the then-if-else construct. If the condition  $c$  is true, the process at the left is executed, otherwise the process at the right-hand side is performed. The language  $\mu\text{CRL}$  also contains the operators  $\underline{\parallel}$ ,  $\mid$  and  $\rho_R$  which play a secondary role, and are not explained here.

Processes are characterized by equations. For instance the process

$$\begin{aligned} \mathbf{proc} \ X(n:\text{Nat}) = & up \cdot X(\text{succ}(n)) + \\ & down \cdot X(\text{pred}(n)) \triangleleft n > 0 \triangleright \delta + \\ & \sum_{m:\text{Nat}} \text{set}(m) \cdot X(m) \end{aligned}$$

denotes a simple counter, that can count up, down and can be reset to a new value  $m$ .

Data are described using equational abstract data types with constructors. To get a flavour, the definition of natural numbers with constructors  $0$  and  $\text{succ}$  with a defined predecessor function is given below.

```
sort Nat
func 0 :  $\rightarrow$  Nat
```

```

    succ : Nat → Nat
map pred : Nat → Nat
var n : Nat
rew pred(succ(n)) = n

```

The advantage of abstract data types is their simplicity and genericity. By employing these a fast and memory efficient toolset for  $\mu\text{CRL}$  has been built [9]. A disadvantage of abstract data types is the need to redefine the basic types for every specification, and the inability to use domain specific technologies, such as for instance integer linear programming.

The language  $\mu\text{CRL}$  has been kept simple. All operators and keywords of the language are given above, except for the keyword **init** used to indicate the initial state of a process.

## Linear processes

Both for manual verification and for tools it turned out to be extremely fruitful to transform all processes to linear form, which in essence is an equation of the form

$$X(\vec{d}; \vec{D}) = \sum_{i \in I} \sum_{\vec{e}_i: \vec{E}_i} a_i(f_i(\vec{d}, \vec{e}_i)) X(g_i(\vec{d}, \vec{e}_i)) \triangleleft c_i(\vec{d}, \vec{e}_i) \triangleright \delta$$

It says that process  $X$  with parameters  $\vec{d}$  can for each  $i \in I$  ( $I$  is a finite index set) choose to perform action  $a_i$  with parameters  $f_i(\vec{d}, \vec{e}_i)$  ending up in process  $X$  with parameters determined by the function  $g_i$ , provided condition  $c_i$  holds. In case a process cannot perform an infinite number of  $\tau$ -actions, it is called  $\tau$ -convergent, or convergent for short.

It is possible to automatically transform any guarded process description to linear form [51]. This includes parallel processes. One of the largest advantages of linear processes is that they do not suffer from the state space explosion problem. Process descriptions of hundreds of pages have been transformed to linear form. Note that linear processes are a common normal form for processes, cf. for instance I/O-automata and the Unity language [14].

## Axioms and rules

Milner [42] provided a concise set of axioms to deal with processes. Nice rules to deal with infinite behaviour were provided by Bergstra and Klop, namely, the recursive specification principle and Koomen's fair abstraction rule (see e.g. [3]). These axioms and rules give a complete underpinning of process algebra.

The recursive specification principle formulated for convergent linear processes equation (CL-RSP) says that every convergent linear process has one solution within the context of branching bisimulation. So, each such linear process exactly defines a process, and moreover, if we can show another process to be a solution of a linear process equation, it must be branching bisimilar to the standard solution of the linear equation.

Koomen’s fair abstraction rule says that every  $\tau$ -loop can be removed from a process. It expresses fairness by saying that if in a  $\tau$ -loop certain actions are iteratively enabled, then one of them must eventually be executed.

As stated before, these elementary rules are too cumbersome to prove complex distributed systems correct. In the next five sections we provide techniques to overcome this.

## Invariants

Remarkably, classic process algebra does not include the notion of an invariant, whereas it is one of the most important concepts in assertional correctness proving. In [6] the notion of an invariant for linear process equations has been introduced. It is a simple predicate that remains true when actions are performed. Formulated in terms of a linear process we say that  $I$  is an invariant if for all  $\vec{d}$ ,  $i \in I$  and  $\vec{e}_i$  it holds that

$$I(\vec{d}) \wedge c_i(\vec{d}, \vec{e}_i) \rightarrow I(g_i(\vec{d}, \vec{e}_i)).$$

This is generally easy to check. A process starting in a state where the invariant holds can be simplified using the invariant.

It turned out that invariants are essential when proving realistic systems correct. But much more is needed.

## Cones and foci

A method that has been totally inspired by the correctness proof of the sliding window protocol is the cones and foci technique [30]. The core difficulty of showing that the sliding window protocol simulates a bidirectional queue lies in the fact that if the queue would perform an action, then the sliding window could only mimic this by first doing a large number of internal steps.

After studying this situation, a rather general pattern emerged in the behaviour of implementations. This can be seen in figure 2. The implementation is generally performing many internal actions to achieve some goal, for instance the delivery of data. This behaviour can be drawn as a cone shaped as a set of states with

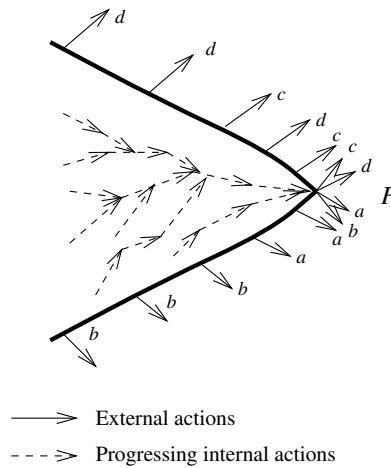


Figure 2: A focus point and its cone

a single state at the focus of the cone. This focus point is where the implementation strives after. All the states in such a cone are bisimilar to a single state in the specification describing the external behaviour of the implementation. An implementation consists of a large number of such cones.

At the focus point, the implementation and the specification can exhibit exactly the same behaviour. But the implementation can already perform external actions, when not at the focus point. E.g. the implementation can already deliver a datum when it still needs to acknowledge it.

Given the fact that implementations contain cones, there is an easy way to prove the implementation and specification weak- or branching bisimilar by providing a *state mapping*  $h$  from the states of an implementation to the states of a specification and by checking the following six properties that can simply be formulated on linear processes:

1. There is no infinite sequence of  $\tau$ -steps. This property can be relaxed by requiring splitting the  $\tau$ -transitions in progressing and non progressing  $\tau$ -transitions, and by requiring that there is no infinite sequence of progressing  $\tau$ 's.
2. The states  $s$  and  $s'$  before and after a  $\tau$ -step, are mapped to the same state in the specification by the state mapping, i.e.  $h(s) = h(s')$ .
3. Each external action that can be done in a state  $s$  in the implementation can also be performed in the related state  $h(s)$  in the specification.
4. For each state  $s$  that is a focus point, each action that can be performed in  $h(s)$  can also be done in  $s$ .

5. Data in related actions in  $s$  and  $h(s)$  must match.
6. The states reached when performing related actions in  $s$  and  $h(s)$  must be related by  $h$  again.

The strength of this technique comes particularly from 4. It is not necessary to check that each action in the specification can be performed in any state in the implementation. It is only necessary to see that such actions can be mimicked in the focus point. The first condition seems particularly restrictive, but it has been relaxed in [20, 30].

There are several generalizations of the cones and foci theorem [20, 22]. The effectiveness of the cones and foci theorem is such that there is now a long list of correctness proofs where this theorem plays a pivotal role.

## Coordinate transformations

Unfortunately, the cones and foci theorem turned out to be insufficient to prove the correctness of the sliding window protocol. Inspired by the work of Anneke Schoone [48], we split the proof in two parts. First we showed that the sliding window protocol with unbounded sequence numbers behaved correctly using the cones and foci method. Then, we showed that this unbounded sliding window protocol behaved the same as the ‘modulo’ sliding window protocol by a simple application of the recursive specification principle. In a sense we went from a non modulo to a modulo coordinate system.

There is a strong resemblance with mathematics, where solving a problem in one coordinate system is much harder than in another coordinate system. Similarly, we believe this is also an essential technique in this field.

Using invariants, cones and foci and coordinate transformations, we were able to give a proof of the sliding window protocol.

## Confluence

Techniques that are totally unrelated to the sliding window protocol, but particularly effective, are  $\tau$ -confluence and  $\tau$ -prioritisation [29]. These techniques are related to partial order reduction but much less involved. In [42] a slightly different notion of confluence had already been mentioned.

The notion of  $\tau$ -confluence (although it comes in many flavours) is best illustrated in a picture (see figure 3). It says that if a process can perform an  $a$  and a  $\tau$  action, it must be able to perform respectively a  $\tau$  and an  $a$  action to a joint state. If this holds for all states and transitions we call the transition system  $\tau$ -confluent.

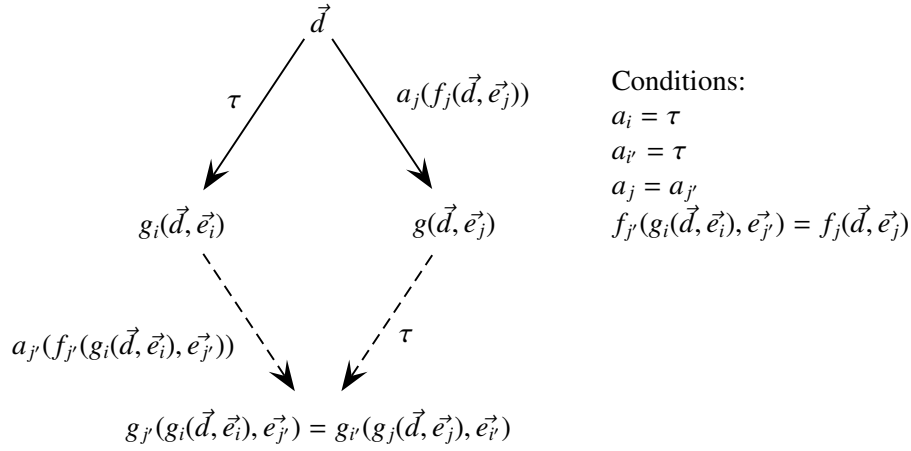


Figure 3: Confluence for a linear process equations

If a system is  $\tau$ -confluent and it is not possible to perform an infinite sequence of  $\tau$ 's, the  $\tau$ -prioritisation operation preserves branching bisimulation. The  $\tau$ -prioritisation operation simply does the following. In any state priority can be given to a  $\tau$ -transition. This means that all other transitions (including other  $\tau$ -transitions) in this state are removed such that only the prioritized  $\tau$  remains.

The use of  $\tau$ -confluence on a given state space is generally less fruitful, as the state space has already been generated. But  $\tau$ -confluence and  $\tau$ -convergence can be established on a linear process. Subsequently,  $\tau$ -prioritisation can be applied to reduce the state space, before it is generated.

In manual correctness proofs,  $\tau$ -confluence can be used to straighten the behaviour of a protocol, for instance by showing that the protocol can be considered as if it operated in several rounds. Without proper foundation such assumptions have been made in the proofs of correctness of several distributed algorithms [16]. When generating state spaces, applying  $\tau$ -confluence and  $\tau$ -prioritisation often reduces its size substantially. There are instances, where the size of the state space after reduction is only the logarithm of the original state space.

## Unbounded parallel processes

Many distributed algorithms deal with an unbounded number of processes. If an individual process  $i$  can be described as  $P(i)$ , the parallel composition of  $n + 1$  such processes is straightforwardly described by the following equation:

$$X(n : Nat) = (P(n) \parallel X(n - 1)) \triangleleft n > 0 \triangleright P(0).$$

It turns out to be non trivial to derive a linear process equation for  $X$ . A way to do this is by inductively adding a single process at the time. The problem is that the behaviour of a restricted number of such processes can be hard to comprehend.

We overcome this problem by a general meta theorem that gives the linear form of  $X$  and that moreover shows that if  $P$  is well defined,  $X$  also uniquely defines a process [31].

## Modal logic

Where process algebra traditionally looks at equating specification and implementation, it is without doubt that establishing the validity of certain properties, neatly formulated using modal formulas, has become an essential means to formulate and establish correctness of systems.

On the one hand the verification of modal formulas has up to now mainly been dominated by providing smart and fast algorithms to establish the validity of such formulas on state spaces. On the other hand a lot of energy went into the fundamental underpinning of the theory. However, relatively little attention has been paid to the development of a mathematical theory to effectively prove the validity of formulas manually. This means that no mathematical experience and subsequent methodology is building up.

Modal formulas must contain data to allow to formulate properties beyond the level that the system contains no deadlock. In [24] it has been pointed out how such enhanced formulas in combination with a linear process can be transformed to parameterized boolean equation systems. Following the line of the excellent thesis by Angelika Mader [39], where the theory for fixed point boolean equation systems is summarized and developed, we are working on extending this theory to solve such equations with data [32, 33].

A typical phenomenon is that some of the equations that we obtain have no easy solutions, and require the investigation of patterns comparable to the patterns occurring in solving differential equations or integrals. We feel that much work needs to be done in this field.

## Toolset

When interested in the specification of correct realistic systems there is one unfortunate observation that one cannot escape. Specifications of realistic systems are large, easily extending dozens of pages. Obviously, only the most extraordinary people can muster the energy to prove properties about objects of this size. In

order to also allow ordinary people to design correct systems, computer tools are required.

For  $\mu$ CRL we have designed a tool primarily centered around the notion of a linear process. In this way we avoid the essential use of automata, which quickly become too large to be handled. As already remarked above, we have transformed large systems of literally hundreds of pages to linear form.

The linear process equation can be transformed and minimised. A simple optimisation is for instance the detection and elimination of parameters that always remain constant and that do not influence external behaviour. Besides  $\tau$ -prioritisation, the elimination of irrelevant parameters is the foremost tool to reduce the size of state spaces. More involved operations are checking and generation of invariants, establishing  $\tau$ -confluence and application of  $\tau$ -prioritisation. Even modal formulas with data can be established on linear process equations where it is of no relevance whether the state space is finite or infinite. The only determining factor is the complexity of the property, the process and in particular the data types.

Currently, after optimising the linear process, we generally resort to the generation of a state transition system and use automata based algorithms for our final analysis. For the generation of the state space several dedicated rewrite technologies have been introduced that allow to calculate with abstract data types almost as fast as with concrete data types. Memory consumption is extremely low by employing sharing offered by the ATerm library [11], leading to a memory consumption of a few bits per parameter in a state for large state spaces. And by employing networks of computers and dedicated algorithms, we can now generate state spaces of more than  $10^9$  states [8] and we expect that around the time this note appears in print it is regularly possible to generate state spaces of  $10^{10}$  states.

The toolset is freely available via [www.cwi.nl/~mcr1](http://www.cwi.nl/~mcr1).

## Open questions

Below we mention a few of the questions that we believe are important to be solved for the further development of this field and into which we already looked to a smaller or larger extent. Of course, the field is broad and therefore much more needs to be done than can possibly be mentioned here.

- Time, probabilities, stochastics and continuous behaviour are all encountered when studying systems. All of these notions have been addressed in many papers, but none of them have a sufficiently simple and mathematically developed form that we know how to incorporate these in our setting. For instance the combination of continuous time, data and branching bisimulation still does not have a satisfying axiomatisation (despite [18, 37, 2]).

- Techniques such as the cones and foci method have been developed to prove branching bisimulation. Sometimes, one encounters specifications and implementations for which equivalences weaker than branching bisimulation are needed. One could think of trace equivalence, failure equivalence or simulation. Within the context of process algebra, no effective means have been developed. An interesting notion could be simulation as used in the context of I/O-automata. But the use of prophecy variables [1] appears to be even more interesting.
- Within distributed systems there are many impossibility results. It is impossible to reach consensus or common knowledge over asynchronous communication channels. Within process algebra, or more generally within the world of behavioural automata there are no nice and general theories to systematically derive and understand these impossibility results.
- Besides process algebra and similar formalisms (such as I/O automata) there are a few rather different formalisms around dealing with the same questions of behaviour and correctness. These are the world of assertional techniques (invariant, weakest preconditions) and Petri-nets. Similar to process algebra, these formalisms have their unique strengths. It is however odd to see that all these domains seem to operate in a rather isolated way, rather than to incorporate each other's techniques and to converge to a unified theory of system behaviour.

For instance, within assertional theorem proving there is a strong but virtually forgotten field of the derivation of algorithms (see e.g. [17]). Based on concise descriptions of the desired effects extremely elegant algorithms have been derived in a very insightful way. Within process algebra such techniques do not exist.

In Petri-nets it is possible to describe distributed data processing effectively. Moreover, it is possible to derive many behavioural properties from the static structure of a Petri-net. Within the context of process algebra similar structural reasoning is largely absent.

## References

- [1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
- [2] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. Monographs in Theoretical Computer Science. Springer Verlag, 2002.

- [3] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge tracts in theoretical computer science 18, Cambridge University Press, 1990.
- [4] J. Berger, P. Middelraad and A.J. Smith. EURIS, European Railway Interlocking Specification. UIC, Commission 7A/16, 1992.
- [5] J.A. Bergstra and J.W. Klop. Fixed point semantics in process algebras. Report IW 206, Mathematisch Centrum, Amsterdam, 1982.
- [6] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings Concur'94*, Uppsala, Sweden, Lecture Notes in Computer Science no. 836, pages 401-416, Springer Verlag, 1994.
- [7] 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.
- [8] S.C.C. Blom. Personal communication, 2003.
- [9] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lissner and J.C. van de Pol.  $\mu$ CRL: A Toolset for Analysing Algebraic Specifications. In proceedings CAV'01. LNCS 2102, pages 250-254, 2001.
- [10] J. Bradfield and C. Stirling. Modal logics and mu-calculi. In J.A. Bergstra, A. Ponse and S.A. Smolka, *Handbook of Process Algebra*, pages 293-332, Elsevier, North-Holland, 2001.
- [11] M.G.J. van den Brand, H.A. de Jong, P. Klint and P.A. Olivier. Efficient Annotated Terms. *Software-Practice and Experience* 30:259-291, 2000.
- [12] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.* C-35 8:677-691, 1986.
- [13] R. Cardell-Oliver. Using higher order logic for modelling real-time protocols. In *Proc. TAPSOFT'91*, LNCS 494, pp. 259–282. Springer, 1991.
- [14] K.M. Chandy and J. Misra. *Parallel Program Design*. A Foundation. Addison Wesley, Reading MA 1988.
- [15] D. Chkhaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *TACAS'03*, LNCS 2619, pp. 113–127. Springer, 2003.
- [16] D. Dolev, M. Klawe and M. Rodeh. An  $O(n \log n)$  unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 4:245-260, 1982.
- [17] W.H.J. Feijen and A.J.M. van Gasteren. *On a method of multiprogramming*. Springer Verlag. 1999.
- [18] W.J. Fokkink. *Clocks, Trees and Stars in Process Theory*. PhD. Thesis. Universiteit van Amsterdam, 1994.
- [19] W.J. Fokkink, J.F. Groote, J. Pang, B. Badban, J.C. van de Pol. Verifying a Sliding Window Protocol in  $\mu$ CRL. Technical report SEN-R0308, CWI, Amsterdam, 2003.

- [20] W.J. Fokkink and J. Pang. Cones and Foci for Protocol Verification Revisited. In Proceedings of 6th Conference on Foundations of Software Science and Computation Structures (FoSSaCS), Lecture Notes in Computer Science 2620, pp. 267-281, Springer-Verlag, 2003
- [21] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM* 43(3):555-600, 1996.
- [22] W.O.D. Griffioen and F.W. Vaandrager. Normed simulations. In A.J.Z. Hu and M.Y. Vardi, editors, Proceedings CAV'98, Vancouver, BC, Canada, LNCS 1427, pages 332-344, Springer-Verlag, 1998.
- [23] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, 110(2):305-326, 1994.
- [24] J.F. Groote and R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In Armando Martin Haebeler, editor, *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology AMAST'99 (Amazonia, Brazil)*, volume 1548 of *Lecture Notes in Computer Science*, pages 74-90. Springer Verlag, January 1999.
- [25] J.F. Groote and H.P. Korver. Correctness proof of the bakery protocol in  $\mu$ CRL. In *Proc. ACP'94*, Workshops in Computing, pp. 63-86. Springer, 1995.
- [26] J.F. Groote and A. Ponse. The syntax and semantics of mCRL. In A. Ponse, C. Verhoef and S.F.M. van Vlijmen, eds, *Algebra of Communicating Processes*, Workshops in Computing, pp. 26-62, 1994.
- [27] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse and S.A. Smolka, *Handbook of Process Algebra*, pages 1151-1208, Elsevier, Amsterdam, 2001.
- [28] W.J. Fokkink, J.F. Groote and M.A. Reniers. Modelling distributed systems. To appear, 2004.
- [29] J.F. Groote and M.P.A. Sellink. Confluence for Process Verification. In *Theoretical Computer Science B (Logic, semantics and theory of programming)*, 170(1-2):47-81, 1996.
- [30] J.F. Groote and J. Springintveld. Focus points and convergent process operators: a proof strategy for protocol verification. *The Journal of Logic and Algebraic Programming* 49, pages 31-60, 2001.
- [31] J.F. Groote and J. van Wamel. The parallel composition of uniform processes with data. *Theoretical Computer Science*, 266:631-6, 2001.
- [32] J.F. Groote and T.A.C. Willemse. A Checker for Modal Formulas for Processes with Data, Eindhoven University of Technology, Department of Computer Science, CSR 02-16, 2002.
- [33] J.F. Groote and T.A.C. Willemse. Parameterized boolean equation systems. Eindhoven University of Technology, Department of Computer Science, In preparation, 2004.

- [34] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
- [35] G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design* 13(3): 289-307, 1998.
- [36] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.
- [37] A.S. Klusener. *Models and axioms for a fragment of real time process algebra*. PhD. Thesis. Eindhoven University of Technology, 1993.
- [38] N. Lynch and M. Tuttle. An Introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1989.
- [39] A. Mader. *Verification of Modal Properties using Boolean Equation Systems*. PhD. Thesis. Fakultät für Informatik. Technische Universität München, Dieter Bertz Verlag, 1997.
- [40] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamentae Informaticae*. XIII:85–139, 1990.
- [41] R. Milner. Processes: 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.
- [42] R. Milner. *A Calculus of Communicating Systems*. volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
- [43] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert. PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference. <http://pvs.csl.sri.com>, 2001.
- [44] K. Paliwoda and J.W. Sanders. An incremental specification of the sliding-window protocol. *Distributed Computing*, 5:83–94, 1991.
- [45] D.M.R. Park. Concurrency and automata on infinite sequences. In *Proceedings of 5th GI Conference*, LNCS 104, pages 167-183. Springer-Verlag, 1981.
- [46] W. Reisig. *Petri Nets*. Springer-Verlag, 1985.
- [47] C. Röckl and J. Esparza. Proof-checking protocols using bisimulations. In *Proc. CONCUR'99*, LNCS 1664, pp. 525–540. Springer, 1999.
- [48] A.A. Schoone. *Protocols by Invariants*. Cambridge International Series on Parallel Computation 7, Cambridge University Press, 1996.
- [49] J.L.A. van de Snepscheut. The sliding window protocol revisited. *Formal Aspects of Computing*, 7(1):3–170, 1995.
- [50] A.S. Tanenbaum. *Computer Networks* (second edition). Prentice-Hall International, 1988.

- [51] Y.S. Usenko. Linearization in  $\mu$ CRL. PhD. Thesis. Eindhoven University of Technology, Eindhoven, 2002.
- [52] F.W. Vaandrager. Verification of two communication protocols by means of process algebra. Report CS-R8608, CWI, Amsterdam, 1986.
- [53] J.J. van Wamel. A study of a one bit sliding window protocol in ACP. Report P9212, University of Amsterdam, 1992.