A Process Algebra Based Verification of a Production System

J.J.T. Kleijn J.E. Rooda
Systems Engineering Group, EUT
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands
j.kleijn@wtb.tue.nl rooda@wtb.tue.nl

M.A. Reniers
Cluster Software Engineering, CWI
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands
Michel.Reniers@cwi.nl

Abstract

Studying industrial systems by simulation enables the designer to study the dynamic behaviour and to determine some characteristics of the system. Unfortunately, simulation also has some disadvantages. These can be overcome by using formal methods. Formal methods allow a thorough analysis of the possible behaviours of a system, parameterised system analysis and a modular approach to the analysis of systems. We present a case study in which a model of an industrial system is studied in a formal way. For this purpose, the model is first specified and simulated using the CSP-based executable specification language χ. The model is translated into a model in the process algebra ACP^{τ}. This enables us to give a correctness proof of the parameterised model and to study the model in isolation.

1. Introduction

Nowadays, industry makes higher demands on methodologies used for the design of new factories. Firstly, due to the huge amount of money involved and growing competition on the market, no mistakes can be afforded. The final design of a factory must be as optimal as possible. Building a new factory or improving a factory step by step is not an option anymore -the new factory should perform optimally from the beginning. Secondly, one must be able to predict system characteristics like throughput and cycle time. This is hampered by the increasing complexity of modern factories. Thirdly, new factories must be realized within shorter periods. Products change faster, new products are developed faster and so must the factories needed to produce them. This requires the possibility of fast and correct modelling of industrial systems.

Modelling and simulation are often used as a helping hand in the design and analysis of industrial systems. This follows from the fact that, usually, models are better suited for experimentation than real systems. Three kinds of models can be distinguished ([12]): physical, graphical, and symbolic. In many applications, symbolic models are easier to construct and manipulate than physical or graphical ones. This is especially true for complex industrial systems. Symbolic models can have a form of verbal descriptions, mathematical expressions, or computer programs. Clearly, it is important to choose a suitable modelling language to construct comprehensible models.

From the point of view of the computational form, symbolic models can be classified as analytical, simulation, and hybrid. Analytical models represented by Markov chains and queueing networks ([3]), although providing fast and precise results, have rather restricted applicability. They can only be used for steady state analysis of relatively simple systems. Simulation models can be applied to a variety of complex dynamic systems. Simulation is widely used for system design and analysis ([6]), but has a drawback of being time consuming and imprecise. Hybrid models are suited for both analytical verification and simulation. Formal specification techniques based on, for example, CSP ([11]), process algebra ([4, 5]) or Petri Nets ([14]) belong to this class of models.

Within the Systems Engineering Group, factories are defined as industrial systems with production units: aggregates of people and machines, groups of machines, or separate machines. Such industrial systems are investigated in order to understand their dynamic behaviour. This is done by modelling these industrial systems using a systems engineering specification formalism called χ ([2, 15]). Within the formalism χ the behaviour of each system component is described by a process using guarded commands ([7]).

Furthermore, individual processes or groups of processes that operate concurrently communicate with each other by means of communication via channels. This aspect of the formalism χ is based on CSP. The latter enables the modeller to introduce non-determinism in his models as it is also present in real-life systems.

Models specified in χ can be simulated ([1]). This proves to be of considerable importance when designing and optimising industrial systems. Namely, the dynamic behaviour of an industrial system can be studied and we can determine its system characteristics (for example, throughput and cycle time versus work in process). Furthermore, a model can easily be adapted to new requirements or specifications. The consequences can be studied by simulating the model again. Also various control strategies can be tested in this way. A lot of money can be saved since changing a model costs only a fraction of making changes to the actual system. Many case studies have already been performed in different industrial areas such as wafer fabs, car assembly systems, fruit juice production and beer breweries.

Unfortunately, simulation does have some disadvantages. Firstly, due to the non-deterministic nature of concurrent systems, one cannot obtain complete information about all possible behaviours of a system by means of simulation. There is always the risk of missing the one behaviour which causes the system to fail. As a consequence, the absence of deadlocks cannot be guaranteed. Secondly, simulation requires the assignment of concrete values to system parameters such as production rates, operating times of machines and the amount of work in process. As a consequence, simulations must be repeated when there is a change in the system parameters. Finally, simulation does not allow a modular analysis of system behaviour. Therefore, it is not possible to study the effects of system changes in isolation. This makes analysis of complex industrial systems very difficult.

Solutions to the problems discussed in the previous paragraph can be found in deriving system characteristics by means of formal methods instead of simulations. Formal methods allow for a thorough analysis of the possible behaviours of a system. For example, one can verify whether or not a system has the desired behaviour. Furthermore, formal methods allow parameterised system analysis. Often, characteristics can be derived for classes of systems. Moreover, using formal methods one can determine the behaviour of parts of the system in isolation, thus allowing a modular approach to the analysis of systems.

In this paper, a case study is presented that deals with the earlier mentioned disadvantages of simulation. The case study considers a production system with an architecture that is found in industry. We derive a χ model of this production system and map this model onto a process algebra model. Firstly, we give a correctness proof of the χ model by verifying the process algebra model using the notion of *focus points and convergent process operators* ([10]). That is, we prove that this model has the desired external behaviour, which among other things, means that it is deadlock free. Secondly, we claim this property for an arbitrary parameter. Thirdly, we study the model in isolation, that is, without considering the environment.

2. The formalism χ

Within the formalism χ, the behaviour of each system component is described by a *process*. These processes can be grouped into a *system* by means of *parallel composition*. Such a system in its turn, can act as a process; it can be combined with other processes and systems to form a new system. In this way, a hierarchical structure can be built in which the resulting top-level system describes the overall behaviour of the modelled system. The relations between processes are defined by *channels*. All channels are one-to-one connections between processes.

To visualise the structure of a model, a graphical representation is used. A process (or system) is represented by a circle with the process name in this circle. A channel is visualised by a curved arrow between the processes that are connected by it. The direction of the arrow indicates the direction of the material or information flow. Somewhere along the arrow, the channel name is given. The graphical representation is not an essential part of the formalism but is helpful in structuring system specifications.

The syntax of the specification language χ is given in BNF notation in Table 1. Note that the syntax of (boolean) expressions is omitted.

Table 1. Syntax of χ statements

x : variable, a : channel, e : expression,
b : boolean expression

PS	$::=$	$x := e$	assignment statement
	$\mid$	$PS;\ PS$	catenation statement
	$\mid$	ES	event statement
	$\mid$	$[\ GP\ ]$	selection statement
	$\mid$	$*[\ GP\ ]$	repetitive selection statement
	$\mid$	$[\ GC\ ]$	selective waiting statement
	$\mid$	$*[\ GC\ ]$	repetitive selective waiting statement
ES	$::=$	$a\,!\,e$	send statement
	$\mid$	$a\,?\,x$	receive statement
GP	$::=$	$b \longrightarrow PS \mid GP\ [\!]\ GP$	
GC	$::=$	$b;\ ES \longrightarrow PS \mid GC\ [\!]\ GC$	

3. The process algebra ACP$^\tau$

The process algebra ACP$^\tau$ is given by its *signature* Σ and a *set of axioms* E. Furthermore, it is parameterised by a set A of *atomic actions* and a partial, commutative and associative *communication function* γ. Atomic actions are the most basic processes considered, and as such we consider them not to be subjected to any further division.

The signature of ACP$^\tau$ consists of a set of *constant* symbols and *function* symbols, often called operators. With these constant and function symbols, *process terms* can be constructed. Firstly, Σ contains atomic actions a of set A. Secondly, it contains the special constants *deadlock* (δ) and *silent step* (τ). When a process is unable to continue or terminate correctly, it yields deadlock. The silent step denotes an unobservable, hidden, action. Thirdly, Σ contains the following operators: the *sequential composition operator* $\cdot$, the *alternative composition operator* $+$, and the *merge operator* $\|$. The *merge operator*, also called *parallel composition operator*, interleaves the behaviours of two processes. However, if two actions can be executed at the same time they may engage in a communication, i.e. a synchronised execution of the two. The result of such a communication is defined by the communication function $\gamma : A \times A \to A$.

The axioms describe when two processes are considered equal. They can be found in [5] (Table 51, page 123).

A semantics is defined using term deduction systems, which consist of a signature and a set of deduction rules, also called Structured Operational Semantics or Plotkin-style semantics. Within these term deduction systems, the following notations are used for $a \in A_\tau$ and $s \in S$ with S the set of terms over the signature: *action step* $\subseteq S \times A \times S$, notation $s \xrightarrow{a} s'$, denotes that s can do an a-step to s' and *action termination* $\subseteq S \times A$, notation $s \xrightarrow{a} \surd$, denotes that s can do an a-step and then terminates.

The semantics of ACP$^\tau$ is given by the term deduction system induced by the deduction rules given in Table 2.

Table 2. Deduction rules for ACP$^\tau$

$$\frac{}{a \xrightarrow{a} \surd} \qquad \frac{x \xrightarrow{a} x'}{x \cdot y \xrightarrow{a} x' \cdot y} \qquad \frac{x \xrightarrow{a} \surd}{x \cdot y \xrightarrow{a} y}$$

$$\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x' \quad y + x \xrightarrow{a} x'} \qquad \frac{x \xrightarrow{a} \surd}{x + y \xrightarrow{a} \surd \quad y + x \xrightarrow{a} \surd}$$

$$\frac{x \xrightarrow{a} x'}{x \| y \xrightarrow{a} x' \| y \quad y \| x \xrightarrow{a} y \| x'} \qquad \frac{x \xrightarrow{a} \surd}{x \| y \xrightarrow{a} y \quad y \| x \xrightarrow{a} y}$$

$$\frac{x \xrightarrow{a} x',\ y \xrightarrow{b} y' \quad \gamma(a,b) = c}{x \| y \xrightarrow{c} x' \| y'} \qquad \frac{x \xrightarrow{a} x',\ y \xrightarrow{b} \surd \quad \gamma(a,b) = c}{x \| y \xrightarrow{c} x',\ y \| x \xrightarrow{c} x'}$$

$$\frac{x \xrightarrow{a} \surd,\ y \xrightarrow{b} \surd,\ \gamma(a,b) = c}{x \| y \xrightarrow{c} \surd}$$

In combination with δ and τ we also use the operators ∂_H and τ_I, representing *encapsulation* and *abstraction*. Encapsulation disables the execution of atomic actions from the set H ($H \subseteq A$). Abstraction turns the atomic actions from the set I ($I \subseteq A$) into unobservable hidden actions. Furthermore, we also introduce the *conditional operator*. This operator enables us to make a distinction between two cases and works as follows: if we consider $x \triangleleft \phi \triangleright y$, then x is executed in the case that the condition ϕ evaluates to *true* and y is executed in the case that ϕ evaluates to *false*. The semantics of the encapsulation, abstraction and conditional operator are given in Table 3.

Consider an implementation of a model that can be described as the parallel composition of several processes $p_1, \ldots, p_n$. These processes communicate via internal actions, where 'internal' means within the system itself and without a direct link to the environment. The resulting internal communications (set I) are renamed as τ. Furthermore, consider a specification *Spec* describing the external behaviour that the implementation should satisfy. In our process algebraic framework, this means that the implementation (after abstraction over the internal actions) should be equal to this specification *Spec* according to some preferred equality relation. In this case, the external behaviour of the implementation and the behaviour of the specification must be *branching bisimilar*.

The basic idea behind the definition of branching bisimulation is that τ-steps can be deleted from computation sequences. A path $t \xrightarrow{\tau}{}^* t' \xrightarrow{a} t''$ in the abstract implementation can be related to a single a-step $s \xrightarrow{a} s'$ in the specification, provided that the sequence of τ-steps $t \xrightarrow{\tau}{}^* t'$ does not yield any relevant change of state.

DEFINITION 3.1 (BRANCHING BISIMULATION) The binary symmetric relation $R \subseteq (S \cup \{\surd\}) \times (S \cup \{\surd\})$ is a branching bisimulation if for all $s, t, s' \in S \cup \{\surd\}$ and $a \in A_\tau$

- if $s \xrightarrow{a} s'$ and $R(s,t)$, then either
    1. $a = \tau$ and $R(s',t)$, or
    2. there exist $t', t'' \in S \cup \{\surd\}$ such that $t \xrightarrow{\tau}{}^* t' \xrightarrow{a} t''$, $R(s,t')$ and $R(s',t'')$,

Table 3. Additional operators

$\dfrac{x \xrightarrow{a} x',\ a \notin H}{\partial_H(x) \xrightarrow{a} \partial_H(x')}$	$\dfrac{x \xrightarrow{a} \surd,\ a \notin H}{\partial_H(x) \xrightarrow{a} \surd}$
$\dfrac{x \xrightarrow{a} x',\ a \notin I}{\tau_I(x) \xrightarrow{a} \tau_I(x')}$	$\dfrac{x \xrightarrow{a} x',\ a \in I}{\tau_I(x) \xrightarrow{\tau} \tau_I(x')}$
$\dfrac{x \xrightarrow{a} \surd,\ a \notin I}{\tau_I(x) \xrightarrow{a} \surd}$	$\dfrac{x \xrightarrow{a} \surd,\ a \in I}{\tau_I(x) \xrightarrow{\tau} \surd}$
$\dfrac{\phi,\ x \xrightarrow{a} x'}{x \lhd \phi \rhd y \xrightarrow{a} x'}$	$\dfrac{\neg\phi,\ y \xrightarrow{a} y'}{x \lhd \phi \rhd y \xrightarrow{a} y'}$
$\dfrac{\phi,\ x \xrightarrow{a} \surd}{x \lhd \phi \rhd y \xrightarrow{a} \surd}$	$\dfrac{\neg\phi,\ y \xrightarrow{a} \surd}{x \lhd \phi \rhd y \xrightarrow{a} \surd}$

- if $s \xrightarrow{a} \surd$ and $R(s,t)$, then either
  1. $a = \tau$ and $t = \surd$, or
  2. there exists $t' \in S \cup \{\surd\}$ such that $t \xrightarrow{\tau}{}^{*} t' \xrightarrow{a} \surd$ and $R(s,t')$.

Two terms $p, q \in S$ are branching bisimilar, notation $p \leftrightarrow_b q$, if there exists a branching bisimulation R such that $R(p,q)$.

We use the notion of focus points and convergent process operators ([10]). It forms a strategy for finding algebraic correctness proofs for communication systems. These communication systems are described in μCRL ([8, 9]), which is roughly ACP$^\tau$ extended with a formal treatment of data.

Proving that two processes are branching bisimilar comes to realising a *state mapping* h that satisfies certain constraints called the *matching criteria*. This state mapping should map states of the abstract implementation to corresponding states of the specification, in such a way that the same set of external actions can be executed directly. Since most of the time states of the abstract implementation do not directly match with a state of the specification, we make an explicit distinction between states in the abstract implementation that do so indeed, and states from which such a state can be reached after some internal computation, i.e. a number of internal steps. States that match directly with a state in the specification are called *focus points*. In such a state, the abstract implementation cannot perform internal actions, that is, there are no outgoing τ-steps. The other states are called *non-focus points*. If the abstract implementation is convergent, we have that from a non-focus point a focus point can be reached by performing finitely many internal actions. Focus points are characterised by the *focus condition*. The set of states that reach the same focus point after some internal activity is called a *cone*. If we have no unbounded internal activity (i.e. no τ-loops exist), every state belongs to some cone. The state mapping now maps all states of a cone to the state corresponding to the focus point of the cone.

Processes are denoted in a particular format called *Deterministic Linear Process Equations* (*D-LPEs*). They enrich the process algebraic language with a symbolic representation of the (possibly infinite) state space of a process by means of state variables and formulas concerning these variables. Furthermore, the booleans are denoted by $\mathbb{B} = \{\top, \bot\}$, where $\top$ denotes true and $\bot$ denotes false. The natural numbers are denoted by $\mathbb{N}$.

DEFINITION 3.2 (*D-LPE*) Let $Act \subseteq A_\tau$ be a finite set of actions. A Deterministic Linear Process Equation (*D-LPE*) over Act and D is an equation of the form

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

for data types E_a, D_a, and functions $f_a : D \to E_a \to D_a$, $g_a : D \to E_a \to D$ and $b_a : D \to E_a \to \mathbb{B}$.

The *D-LPE* p in the above definition describes that process p in a state described by d can perform the action $a(f_a(d,e_a))$ if for some e_a of type E_a the guard $b_a(d,e_a)$ is satisfied. After the execution the state of process p is described by $g_a(d,e_a)$. In a *D-LPE* there is at most one summand in the alternative composition for every $a \in Act$.

In the sequel, when we write $f_a(d,e_a)$, $g_a(d,e_a)$ and $b_a(d,e_a)$, we refer to the abstract implementation, whereas with $f'_a(d,e_a)$, $g'_a(d,e_a)$ and $b'_a(d,e_a)$, we refer to the specification.

DEFINITION 3.3 (FOCUS CONDITION) The focus condition $FC_p(d)$ of *D-LPE* p in state d is the formula

$$\neg\exists_{e_\tau : E_\tau}(b_\tau(d,e_\tau)).$$

DEFINITION 3.4 (MATCHING CRITERIA) Let $p(d{:}D)$ and $q(d'{:}D')$ be *D-LPE*s. Then the function $h : D \to D'$ is a state mapping if it satisfies the following *matching criteria* for all $e_\tau{:}E_\tau$, $a \in Act \setminus \{\tau\}$, and $e_a{:}E_a$

$$p \text{ is convergent}, \quad (1)$$

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

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

$$FC_p(d) \wedge b'_a(h(d), e_a) \Rightarrow b_a(d, e_a), \quad (4)$$

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

$$b_a(d, e_a) \Rightarrow h(g_a(d, e_a)) = g'_a(h(d), e_a). \quad (6)$$

In [10], the matching criteria are explained as follows: Criterion (1) says that p must be convergent. This means that in a cone every internal action τ must constitute progress towards a focus point. Criterion (2) says that if in a state d in the abstract implementation an internal step can be done ($b_\tau(d, e_\tau)$ is valid), then this internal step is not observable. This is described by saying that both states relate to the same state in the specification. Criterion (3) says that when the abstract 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 abstract implementation such that $h(d) = e$ an a-step can be done *after* some internal actions. The latter is guaranteed by criterion (4). This criterion says that in a focus point of the abstract implementation, an action a 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, respectively.

THEOREM 3.1 Let $p(d{:}D)$ and $q(d'{:}D')$ be *D-LPE*s and $h : D \to D'$ a state mapping. If q does not contain τ-steps, then p and q are branching bisimilar: $p \leftrightarrow_b q$.

4. The case study

This particular case study deals with a production system. It consists of a stocker S, a transporting mechanism T, a machine M and a controller C. This is depicted in Figure 1. The stocker obtains products from the environment via channel es. These products are transported to machine M through the channels st and tm. Machine M collects the incoming products until the amount of collected products has reached the batch size (s). At that moment, the newly formed batch is sent back to S via the transporter through the channels mt and ts. S collects those batches and returns them to the environment via channel se. The interaction between the transporting mechanism, which can only handle one product or batch at a time, and the machine is controlled by process C. It exchanges information with T and M via the channels ct and cm respectively.

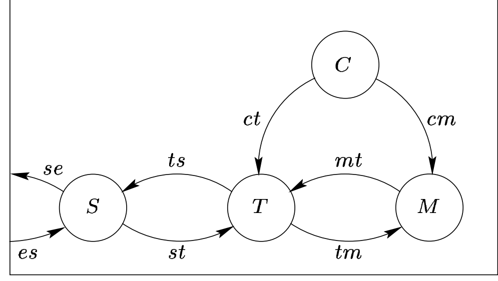

Figure 1. Production system

Firstly, we specify the χ models of the specification (Ψ) and implementation (Φ) of the production system. From these χ models we derive the *D-LPE* models Ψ and Ξ, the latter being the abstract implementation. In Section 5, we then realise a state mapping h, which maps the states of Ξ to the corresponding states of Ψ, and prove that $\Xi \leftrightarrow_b \Psi$.

Within the specification language χ, we can abbreviate an expression '*true*; ES' to ES. This is done in the process descriptions of Ψ and S. Furthermore, products are represented by naturals. Batches are lists (notation: *type**) of products. In χ the types 'prod' and 'batch' are then declared as follows.

type $\text{prod} = \text{nat}$
$\quad , \ \text{batch} = \text{prod}^*$

Before we present the χ models, the used predefined functions *take* and *drop* are given. Furthermore, the constant $[\,]$ denotes the empty list and the function $+\!\!+$ denotes concatenation of lists. The function *len* gives the length of a list and the functions *hd* and *tl* give the head and the tail of a list, respectively.

DEFINITION 4.1 Let T be an arbitrary data type. Then for all $x \in T$, $xs \in T^*$ and $n \in \mathbb{N}$ the functions $take, drop : T^* \times \mathbb{N} \to T^*$ are defined by

$$\begin{aligned} take([\,], n) &= [\,], \\ take(xs, 0) &= [\,], \\ take([x] +\!\!+ xs, n+1) &= [x] +\!\!+ take(xs, n), \\ drop([\,], n) &= [\,], \\ drop(xs, 0) &= xs, \\ drop([x] +\!\!+ xs, n+1) &= drop(xs, n). \end{aligned}$$

The desired external behaviour, to be specified in process Ψ, can be described as follows: products enter the production system via channel es. Every time process Ψ contains a number of products equal to or greater than the batch size s, a batch can be returned to the environment via channel se.

```
proc Ψ(es : ?prod, se : !batch, s : nat) =
⟦ x : prod, xs : batch | xs := []
; *[                     es ? x          ⟶ xs := xs ++ [x]
  ▯ len(xs) ≥ s;  se ! take(xs, s) ⟶ xs := drop(xs, s)
  ]
⟧
```

The implementation Φ consists of the parallel composition of the processes S, T, M and C.

```
proc S(es:?prod, st:!prod, ts:?batch, se:!batch) =
⟦ x: prod, xs, y: batch, ys: batch* | xs := []; ys := []
; *[                 es ? x      ⟶ xs := xs ++ [x]
  ▯ len(xs) > 0; st ! hd(xs) ⟶ xs := tl(xs)
  ▯                  ts ? y      ⟶ ys := ys ++ [y]
  ▯ len(ys) > 0; se ! hd(ys) ⟶ ys := tl(ys)
  ]
⟧
```

```
proc T( st : ?prod, tm : !prod, mt : ?batch
      , ts : !batch, ct : ?bool) =
⟦ x : prod, xs : batch, z : bool | xs := []; z := true
; *[ [   z ; st ? x    ⟶ xs := xs ++ [x]; tm ! hd(xs)
                          ; xs := []
     ▯ ¬ z ; mt ? xs ⟶ ts ! xs; xs := []
     ]
   ; ct ? z
   ]
⟧
```

```
proc M(tm : ?prod, mt : !batch, cm : ?bool) =
⟦ x : prod, xs : batch, z : bool | xs := []; z := true
; *[ [   z ; tm ? x  ⟶ xs := xs ++ [x]
     ▯ ¬ z ; mt ! xs ⟶ xs := []
     ]
   ; cm ? z
   ]
⟧
```

```
proc C(ct, cm : !bool, s : nat) =
⟦ i : nat | i := 0
; *[ ct !(i + 1 ≠ s); cm !(i + 1 ≠ s)
   ; i := (i + 1) mod (s + 1)
   ]
⟧
```

```
syst Φ(es : ?prod, se : !batch, s : nat) =
⟦  st, tm : prod, mt, ts : batch, ct, cm : bool
| S(es, st, ts, se) ∥ T(st, tm, mt, ts, ct)
∥ M(tm, mt, cm) ∥ C(ct, cm, s)
⟧
```

Let us now specify the *D-LPE* models for the specification and the abstract implementation. Deriving them from the corresponding χ models is straightforward.

Because the *D-LPE* format forces us to maintain former sequential compositions within a process by means of data manipulation, extra state parameters $j_t, j_m, j_c \in I\!N$ are introduced in the processes T, M and C. Furthermore, in order to increase readability of the expressions to follow, we will only show those parameters of a process in its parameter list that are actually modified. If a variable v is replaced by an expression e, then this is denoted by e/v.

To facilitate the verification in Section 5, we assume that consecutive products entering our system are also labelled consecutively, starting at 1. Therefore, we introduce the parameters x_Ψ and x_s in the processes Ψ and S. They represent the next product to be received via channel es.

Let us now consider the specification. For the specification the parameters are a list of products xs_Ψ present in the system, the batch size s, and the next product x_Ψ to be received. The state space of the specification is then given by $D_\Psi = I\!N \times I\!N^* \times I\!N^{>0}$.

DEFINITION 4.2 The specification is given by the process $\Psi(1, [\,], s)$, where

$$\begin{aligned} &\Psi(x_\Psi, xs_\Psi, s)\\ &= r_{es}(x_\Psi) \cdot \Psi(x_\Psi + 1/x_\Psi, xs_\Psi +\!\!+ [x_\Psi]/xs_\Psi)\\ &+ s_{se}(take(xs_\Psi, s)) \cdot \Psi(drop(xs_\Psi, s)/xs_\Psi)\\ &\quad \lhd len(xs_\Psi) \geq s \rhd \delta. \end{aligned}$$

Before we construct the model for the abstract implementation, we first specify its components. The components are the processes S, T, M and C.

DEFINITION 4.3 For $(x_s, xs_s, ys_s) \in D_S$, where $D_S = I\!N \times I\!N^* \times (I\!N^*)^*$, we define

$$\begin{aligned} &S(x_s, xs_s, ys_s)\\ &= r_{es}(x_s) \cdot S(x_s + 1/x_s, xs_s +\!\!+ [x_s]/xs_s)\\ &+ s_{st}(hd(xs_s)) \cdot S(tl(xs_s)/xs_s) \lhd len(xs_s) > 0 \rhd \delta\\ &+ \sum_{y:I\!N^*} r_{ts}(y) \cdot S(ys_s +\!\!+ [y]/ys_s)\\ &+ s_{se}(hd(ys_s)) \cdot S(tl(ys_s)/ys_s) \lhd len(ys_s) > 0 \rhd \delta. \end{aligned}$$

For $(xs_t, z_t, j_t) \in D_T$, where $D_T = I\!N^* \times I\!B \times I\!N$, we define

$$\begin{aligned} &T(xs_t, z_t, j_t)\\ &= \sum_{x:I\!N} r_{st}(x) \cdot T(xs_t +\!\!+ [x]/xs_t, 1/j_t) \lhd z_t \wedge j_t = 0 \rhd \delta\\ &+ s_{tm}(hd(xs_t)) \cdot T([\,]/xs_t, 2/j_t) \lhd z_t \wedge j_t = 1 \rhd \delta\\ &+ \sum_{xs:I\!N^*} r_{mt}(xs) \cdot T(xs_t +\!\!+ xs/xs_t, 1/j_t)\\ &\quad \lhd \neg z_t \wedge j_t = 0 \rhd \delta\\ &+ s_{ts}(xs_t) \cdot T([\,]/xs_t, 2/j_t) \lhd \neg z_t \wedge j_t = 1 \rhd \delta\\ &+ \sum_{z:I\!B} r_{ct}(z) \cdot T(z/z_t, 0/j_t) \lhd j_t = 2 \rhd \delta. \end{aligned}$$

For $(xs_m, z_m, j_m) \in D_M$, where $D_M = I\!N^* \times I\!B \times I\!N$, we define

$$
\begin{aligned}
&M(xs_m, z_m, j_m)\\
&= \sum_{x:\mathbb{N}} r_{tm}(x) \cdot M(xs_m +\!\!+ [x]/xs_m, 1/j_m)\\
&\quad \lhd z_m \wedge j_m = 0 \rhd \delta\\
&+ s_{mt}(xs_m) \cdot M([]/xs_m, 1/j_m) \lhd \neg z_m \wedge j_m = 0 \rhd \delta\\
&+ \sum_{z:\mathbb{B}} r_{cm}(z) \cdot M(z/z_m, 0/j_m) \lhd j_m = 1 \rhd \delta.
\end{aligned}
$$

For $(i_c, j_c, s) \in D_C$, where $D_C = \mathbb{N} \times \mathbb{N} \times \mathbb{N}^{>0}$, we define

$$
\begin{aligned}
&C(i_c, j_c, s)\\
&= s_{ct}(i_c + 1 \neq s) \cdot C(1/j_c) \lhd j_c = 0 \rhd \delta\\
&+ s_{cm}(i_c + 1 \neq s) \cdot C((i_c + 1) \bmod (s+1)/i_c, 0/j_c)\\
&\quad \lhd j_c = 1 \rhd \delta.
\end{aligned}
$$

These processes communicate over the channels as depicted in Figure 1. For this case study, γ is, for $ch \in \mathcal{C}$ and $d \in \mathcal{D}$, the universes of all channel names and all data values respectively, defined by

$$\gamma(s_{ch}(d), r_{ch}(d)) = c_{ch}(d).$$

The implementation is obtained by putting the individual processes S, T, M and C in parallel and disabling the occurrence of the loose send and receive actions. This is achieved by means of the encapsulation operator ∂_H, where

$$
\begin{aligned}
H = &\{s_{st}(d), r_{st}(d), s_{tm}(d), r_{tm}(d) \mid d \in \mathbb{N}\}\\
\cup\ &\{s_{mt}(d), r_{mt}(d), s_{ts}(d), r_{ts}(d) \mid d \in \mathbb{N}^*\}\\
\cup\ &\{s_{ct}(d), r_{ct}(d), s_{cm}(d), r_{cm}(d) \mid d \in \mathbb{B}\}.
\end{aligned}
$$

The implementation can then be described by

$$\partial_H(S(1, [], []) \parallel T([], \top, 0) \parallel M([], \top, 0) \parallel C(0, 0, s)).$$

Then, we abstract from the internal actions of the implementation to arrive at an abstract implementation. The internal actions, i.e. successful communications over the internal channels, are given by the set

$$
\begin{aligned}
I = &\{c_{st}(d), c_{tm}(d) \mid d \in \mathbb{N}\}\\
\cup\ &\{c_{mt}(d), c_{ts}(d) \mid d \in \mathbb{N}^*\}\\
\cup\ &\{c_{ct}(d), c_{cm}(d) \mid d \in \mathbb{B}\}.
\end{aligned}
$$

This abstraction is achieved by applying the abstraction operator τ_I. The abstract implementation is then given by

$$\tau_I(\partial_H(S(1,[],[]) \parallel T([],\top,0) \parallel M([],\top,0) \parallel C(0,0,s))).$$

The state space of the abstract implementation can be seen as a compound state space of the state spaces of the individual processes. It is denoted by $D_\Xi = D_S \times D_T \times D_M \times D_C$. Next, we linearise the abstract implementation, and the result is referred to as Ξ in the sequel. The τ-actions are labelled with their corresponding channel name. This shows from which internal communication action the τ originates and is useful for referring purposes.

THEOREM 4.1 The abstract implementation Ξ is the process $\Xi(1, [], [], [], \top, 0, [], \top, 0, 0, 0, s)$, where

$$
\begin{aligned}
&\Xi(x_s, xs_s, ys_s, xs_t, z_t, j_t, xs_m, z_m, j_m, i_c, j_c, s)\\
&= r_{es}(x_s) \cdot \Xi(x_s + 1/x_s, xs_s +\!\!+ [x_s]/xs_s)\\
&+ \tau_{st} \cdot \Xi(tl(xs_s)/xs_s, xs_t +\!\!+ [hd(xs_s)]/xs_t, 1/j_t)\\
&\quad \lhd len(xs_s) > 0 \wedge z_t \wedge j_t = 0 \rhd \delta\\
&+ \tau_{tm} \cdot \Xi([]/xs_t, 2/j_t, xs_m +\!\!+ [hd(xs_t)]/xs_m, 1/j_m)\\
&\quad \lhd z_t \wedge j_t = 1 \wedge z_m \wedge j_m = 0 \rhd \delta\\
&+ \tau_{mt} \cdot \Xi(xs_t +\!\!+ xs_m/xs_t, 1/j_t, []/xs_m, 1/j_m)\\
&\quad \lhd \neg z_t \wedge j_t = 0 \wedge \neg z_m \wedge j_m = 0 \rhd \delta\\
&+ \tau_{ts} \cdot \Xi(ys_s +\!\!+ [xs_t]/ys_s, []/xs_t, 2/]j_t)\\
&\quad \lhd \neg z_t \wedge j_t = 1 \rhd \delta\\
&+ s_{se}(hd(ys_s)) \cdot \Xi(tl(ys_s)/ys_s) \lhd len(ys_s) > 0 \rhd \delta\\
&+ \tau_{ct} \cdot \Xi(i_c < s/z_t, 0/j_t, 1/j_c) \lhd j_t = 2 \wedge j_c = 0 \rhd \delta\\
&+ \tau_{cm} \cdot \Xi(i_c < s/z_m, 0/j_m, (i_c+1) \bmod (s+1)/i_c, 0/j_c)\\
&\quad \lhd j_m = 1 \wedge j_c = 1 \rhd \delta.
\end{aligned}
$$

5. Verification

In this section we define a state mapping $h : D_\Xi \to D_\Psi$ between the abstract implementation and the specification. This mapping relates states from the abstract implementation to states from the specification.

In this case study, the mapping h is defined in two steps. Since the products that enter the production system never overtake and the products are numbered consecutively, the state can be described in terms of the minimal and maximal product number present in the system. A transformation of the state of the abstract implementation into these two numbers is defined by the function $h_1 : D_\Xi \to D_{\Xi'}$, where $D_{\Xi'} = \mathbb{N} \times \mathbb{N}$, the minimal and maximal product number.

In the specification the state is described in terms of a list of the product numbers that are in the system. Again, under the assumption that the products are numbered consecutively, it is not hard to obtain this list once the minimal and maximal product number are known. This is achieved by the auxiliary function $h_2 : D_{\Xi'} \to D_\Psi$.

Let us look at function h_1 in more detail. It is defined in terms of the function *min*, which returns the minimal product number present in the system and the function *max*, which returns the maximal product number present in the system. We find the minimal product number present in the system by walking through the system in reverse until a product is encountered. If our system is empty, the minimal product number equals x_s by definition. The maximal product number equals $x_s - 1$ (Proposition 5.1).

Before we define the functions *min* and *max*, let us first define the functions *flat* and *list*. Function *flat* flattens a list of lists and function *list* creates a list containing the products present in the system.

In the sequel, we use $d \in D_\Xi$ as a shorthand notation for the list of parameters of the abstract implementation: $(x_s, xs_s, ys_s, xs_t, z_t, j_t, xs_m, z_m, j_m, i_c, j_c, s)$. The

initial configuration of this list in the abstract implementation, $(1, [\,], [\,], [\,], \top, 0, [\,], \top, 0, 0, 0, s)$, will be referred to as d_0, whereas a new configuration after an arbitrary action originating from a configuration d will be referred to as $d^\star$.

DEFINITION 5.1 Let T be an arbitrary data type. Then for all $xs : T^*$, $xss : (T^*)^*$ the function *flat* : $(T^*)^* \to T^*$ is defined by

$$\begin{aligned} \mathit{flat}([\,]) &= [\,], \\ \mathit{flat}([xs] \mathbin{+\!\!+} xss) &= xs \mathbin{+\!\!+} \mathit{flat}(xss). \end{aligned}$$

The function *list* : $D_\Xi \to \mathbb{N}^*$ is defined by

$$\begin{aligned} \mathit{list}(d) = &\mathit{flat}(ys_s) \\ &\mathbin{+\!\!+}((xs_t \mathbin{+\!\!+} xs_m) \triangleleft \neg z_t \triangleright (xs_m \mathbin{+\!\!+} xs_t)) \\ &\mathbin{+\!\!+} xs_s \mathbin{+\!\!+} [x_s]. \end{aligned}$$

The functions *min*, *max* : $D_\Xi \to \mathbb{N}$ are defined by

$$\begin{aligned} \mathit{min}(d) &= \mathit{hd}(\mathit{list}(d)), \\ \mathit{max}(d) &= (x_s - 1) \max 0. \end{aligned}$$

The function h_2 is defined in terms of the function *np*, which returns the number of the next product to be received via channel es (x_s), and the function *pp*, which constructs the list of present products (xs_Ψ).

DEFINITION 5.2 The functions *np* : $D_{\Xi'} \to \mathbb{N}$ and *pp* : $D_{\Xi'} \to \mathbb{N}^*$ are defined by

$$\begin{aligned} \mathit{np}(m, n) &= n + 1, \\ \mathit{pp}(m, n) &= [m, \ldots, n], \end{aligned}$$

where $[m, \ldots, n]$ denotes the empty list $[\,]$ in the case that $m > n$ and otherwise denotes a list of consecutive numbers from m to n.

DEFINITION 5.3 The functions $h_1 : D_\Xi \to D_{\Xi'}$, $h_2 : D_{\Xi'} \to D_\Psi$, and $h : D_\Xi \to D_\Psi$ are defined by

$$\begin{aligned} h_1(d) &= (\mathit{min}(d), \mathit{max}(d)), \\ h_2(d) &= (\mathit{np}(d'), \mathit{pp}(d')), \\ h(d) &= h_2(h_1(d)). \end{aligned}$$

PROPOSITION 5.1 $\Xi(d_0) \vdash \mathit{max}(d) = x_s - 1$.

PROOF. For all reachable states of our abstract implementation Ξ (Theorem 4.1), it is derivable that $x_s > 0$, or in shorthand notation: $\Xi(d_0) \vdash x_s > 0$. Furthermore, since $\Xi(d_0) \vdash x_s > 0$, we obtain $\Xi(d_0) \vdash \mathit{max}(d) = x_s - 1$. □

Next, we show that the state mapping h satisfies the matching criteria, which implies that specification and abstract implementation of our production system are branching bisimilar (Theorem 3.1). This is done by discussing the criteria one by one. For the most interesting criterion, criterion (6), the proof is provided. The proofs of the other criteria can be found in [13].

Criterion (1) says that Ξ must be convergent. In case of our production system, we have that Ξ does not contain any infinite internal computations. Thus, Ξ is convergent.

Criterion (2) says that if in a state d in the abstract implementation an internal step can be done ($b_\tau(d, e_\tau)$ is valid), then this internal step is not observable. This is described by saying that both states relate to the same state in the specification. In our case, internal steps represent internal movement of material or transport of information. None of these actions have an effect on the number of products present in the system nor do they effect their order.

Criterion (3) says that when the abstract implementation can perform an external step, then the corresponding state in the specification must also be able to perform this step. In case of our production system, we have that both specification and abstract implementation can always receive a new product. Furthermore, we can prove that when $\mathit{len}(ys_s) > 0$ we also have $\mathit{len}(xs_\Psi) \geq s$. This means that when the abstract implementation is able to deliver a completed batch, the specification is able to do the same.

Criterion (4) says that in a focus point of the abstract implementation, an action a in the abstract implementation can be performed if it is enabled in the specification. In our case, it holds that both the specification and the abstract implementation can receive a new product. Concerning the delivering of processed batches, we can prove that when $\mathit{len}(xs_\Psi) \geq s$, we can also realize $\mathit{len}(ys_s) > 0$, the possibility to deliver a completed batch in the abstract implementation, provided that no τ step is enabled.

Criterion (5) says that corresponding external actions carry the same data parameter (modulo h).

Criterion (6) says that corresponding external actions lead to corresponding states. We prove that this is the case for our abstract implementation and specification.

PROOF. Proving that the function h satisfies criterion (6) means proving that it holds for any external action.

Let us first consider the action $r_{es}(x_s)$. In that case we have $b_a(d)$ and $g_a(d) = d[x_s + 1/x_s,\ xs_s \mathbin{+\!\!+} [x_s]/xs_s]$. The latter is denoted as $d^\star$ in the sequel. We have

$$\begin{aligned} h(d^\star) &= (\mathit{np}(\mathit{min}(d^\star), \mathit{max}(d^\star)), \mathit{pp}(\mathit{min}(d^\star), \mathit{max}(d^\star))), \\ g'_a(h(d)) &= (\mathit{np}(\mathit{min}(d), \mathit{max}(d)) + 1 \\ &\quad ,\mathit{pp}(\mathit{min}(d),\mathit{max}(d)) \mathbin{+\!\!+} [\mathit{np}(\mathit{min}(d),\mathit{max}(d))]). \end{aligned}$$

Then, the following propositions remain to prove

$$\mathit{np}(\mathit{min}(d^\star), \mathit{max}(d^\star)) = \mathit{np}(\mathit{min}(d), \mathit{max}(d)) + 1, \quad (1)$$

$$\begin{aligned} \mathit{pp}(\mathit{min}(d^\star), \mathit{max}(d^\star)) = &\,\mathit{pp}(\mathit{min}(d), \mathit{max}(d)) \\ &\mathbin{+\!\!+} [\mathit{np}(\mathit{min}(d), \mathit{max}(d))]. \end{aligned} \quad (2)$$

Before we prove these propositions, let us first prove that in general, for $r_{es}(x_s)$, the following two propositions hold

$$min(d^\star) = min(d), \tag{3}$$
$$max(d^\star) = max(d) + 1. \tag{4}$$

For the proof of (3), consider the following computation

$$\begin{aligned} & min(d^\star) \\ =& hd(list(d^\star)) \\ =& hd(flat(ys_s) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} (xs_s \mathbin{+\!\!+} [x_s]) \mathbin{+\!\!+} [x_s + 1]) \\ =& hd(flat(ys_s) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} xs_s \mathbin{+\!\!+} [x_s]) \\ =& min(d). \end{aligned}$$

For the proof of (4), consider the following computation

$$\begin{aligned} & max(d^\star) \\ =& max(d[x_s + 1/x_s,\ xs_s \mathbin{+\!\!+} [x_s]/xs_s]) \\ =& (x_s + 1) - 1 \\ =& x_s \\ =& max(d) + 1. \end{aligned}$$

Now these propositions are proved, we can use them while proving (1) and (2). For the proof of (1), consider the following computation

$$\begin{aligned} & np(min(d^\star), max(d^\star)) \\ =& max(d^\star) + 1 \\ =& (max(d) + 1) + 1 \\ =& np(min(d), max(d)) + 1. \end{aligned}$$

For the proof of (2), consider the following computation

$$\begin{aligned} & pp(min(d^\star), max(d^\star)) \\ =& [min(d^\star), \ldots, max(d^\star)] \\ =& [min(d), \ldots, max(d) + 1] \\ =& [min(d), \ldots, max(d)] \mathbin{+\!\!+} [max(d) + 1] \\ =& pp(min(d), max(d)) \mathbin{+\!\!+} [np(min(d), max(d))]. \end{aligned}$$

Let us now consider the action $s_{se}(hd(ys_s))$. In that case, $b_a(d) = len(ys_s) > 0$ and $g_a(d) = d[tl(ys_s)/ys_s]$. Again, we use the shorthand notation $d^\star$ for $d[tl(ys_s)/ys_s]$.

We have

$$\begin{aligned} h(d^\star) =& (np(min(d^\star), max(d^\star)), pp(min(d^\star), max(d^\star))), \\ g'_a(h(d)) =& (np(min(d), max(d)) \\ & ,drop(pp(min(d), max(d)), s)). \end{aligned}$$

In that case, the following propositions remain to prove

$$np(min(d^\star), max(d^\star)) = np(min(d), max(d)), \tag{5}$$
$$pp(min(d^\star), max(d^\star)) = drop(pp(min(d), max(d)), s). \tag{6}$$

Before we prove these propositions, we first prove that in general, for $s_{se}(hd(ys_s))$, the following propositions hold

$$min(d^\star) = min(d) + s, \tag{7}$$
$$max(d^\star) = max(d). \tag{8}$$

For the proof of (7), consider the following computation

$$\begin{aligned} & min(d^\star) \\ =& hd(list(d^\star)) \\ =& hd(flat(tl(ys_s)) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} [x_s]) \\ =& hd(drop(flat(ys_s), s) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} [x_s]) \\ =& hd(flat(ys_s) \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} [x_s]) + s \\ =& hd(list(d)) + s \\ =& min(d) + s. \end{aligned}$$

For the proof of (8), consider the following computation

$$\begin{aligned} & max(d^\star) \\ =& max(d[tl(ys_s)/ys_s]) \\ =& x_s - 1 \\ =& max(d). \end{aligned}$$

Now these propositions are proved, let us prove (5) and (6). For the proof of (5), consider the following computation

$$\begin{aligned} & np(min(d^\star), max(d^\star)) \\ =& max(d^\star) + 1 \\ =& max(d) + 1 \\ =& np(min(d), max(d)). \end{aligned}$$

For the proof of (6), consider the following computation

$$\begin{aligned} & pp(min(d^\star), max(d^\star)) \\ =& [min(d^\star), \ldots, max(d^\star)] \\ =& [min(d) + s, \ldots, max(d)] \\ =& drop([min(d), \ldots, max(d)], s) \\ =& drop(pp(min(d), max(d)), s). \end{aligned}$$

This completes the proof of criterion (6). □

THEOREM 5.1 For any batch size s, we have $\Xi \leftrightarrow_b \Psi$.

PROOF. This follows immediately from Theorem 3.1 and the fact that h, as defined in Definition 5.3, is a state mapping (satisfies the matching criteria). For complete proofs we refer to [13]. □

6. Conclusions

We investigated whether formal methods could be used for the analysis of industrial systems models specified using the specification language χ. The case study described

in this paper was a first attempt. It dealt with a model of an industrial system and had three objectives. The first objective was to give a correctness proof of this model. The second objective was to determine system properties for arbitrary parameters and the third objective was to study the model in isolation. The objectives are met through the use of formal methods.

Before we draw conclusions concerning these matters, let us first conclude the following on the use of χ, ACP$^\tau$, and focus points and convergent process operators:

- Using the formalism χ, one can specify real-life models quickly and easily. These models can then be validated by means of simulation. In that way, results can be obtained within a short period of time. Specifying formal models, like ACP$^\tau$ models, is more difficult and time consuming. Furthermore, for the time being, verification of such models is restricted to models of a small size. On the other hand, formal models can be verified, whereas χ models can only be validated.
- The biggest change to be made when mapping a χ model onto an ACP$^\tau$ model is that one has to develop a specification using a procedural formalism, where parameters can only change when processes get invoked, originating from a model specified in a state oriented formalism working with assignments.
- Focus points and convergent process operators are very well suited for the actual verification of the model since they provide a strategy for finding algebraic correctness proofs for communication systems.
- The efficacy of the state mapping between the abstract implementation and the specification for a great deal determines the burden of the proof to follow.

Since we verified the ACP$^\tau$ model and because we may, on intuitive grounds, conclude that we succeeded in preserving the semantics while deriving the ACP$^\tau$ model from the χ model, we are allowed to apply the statements to be made on the ACP$^\tau$ model to the χ model. Current research is concerned with enabling direct formal reasoning about χ models by setting up a formal semantics of χ.

We succeeded in giving a correctness proof by proving that the external behaviour of the implementation is branching bisimilar with the specification ($\Xi \leftrightarrow_b \Psi$). Moreover, we proved the model to be correct for an arbitrary batch size s. Finally, we were able to study the model in isolation. Unlike in simulation, no addition of processes describing the environment is needed. This allows a modular approach to the analysis of systems.

Our first step in integrating formal methods with an existing design methodology for industrial systems proved to be successful. Future research will be concerned with performing more extensive case studies in industry and the development of required tool support. The latter should enable verification of real-life models. Furthermore, it should decrease the time needed to come to correct design of industrial systems.

ACKNOWLEDGEMENTS Thanks go to J.C.M. Baeten, J.F. Groote, J.M. van de Mortel-Fronczak, S. Mauw, and M. van der Zwaag for their valuable remarks on preliminary (and extended) versions of this paper.

References

[1] W. Alberts and G. Naumoski. *A Discrete-Event Simulator for Systems Engineering*. PhD thesis, Eindhoven University of Technology, 1998.

[2] N. Arends. *A Systems Engineering Specification Formalism*. PhD thesis, Eindhoven University of Technology, 1996.

[3] R. Askin and C. Standridge. *Modeling and Analysis of Manufacturing systems*. John Wiley & Sons, 1993.

[4] J. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, Semantic Modelling, pages 149–268. Oxford University Press, 1995.

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

[6] A. Carrie. *Simulation of Manufacturing Systems*. John Wiley & Sons, 1988.

[7] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.

[8] J. Groote and A. Ponse. Proof theory for *μCRL*: a language for processes with data. In *Proceedings of the International Workshop on Semantics of Specification Languages*, Workshops in Computing, pages 231–250. Springer-Verlag, 1994.

[9] J. Groote and A. Ponse. The syntax and semantics of *μCRL*. In *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62. Springer-Verlag, 1994.

[10] J. Groote and J. Springintveld. Focus points and convergent process operators: A proof strategy for protocol verification. Logic Group Preprint Series 142, Utrecht Research Institute for Philosophy, Oct. 1995.

[11] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] B. Khoshnevis. *Discrete Systems Simulation*. McGraw-Hill, 1994.

[13] J. Kleijn and J. Rooda. Verifying the correctness of industrial systems models, a case study using a systems engineering specification formalism and analytical methods. Technical Report SE 420175, Eindhoven University of Technology, 1998.

[14] J. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, 1981.

[15] J. van de Mortel-Fronczak and J. Rooda. Heterarchical control systems for production cells — a case study. In *Proceedings of MIM'97*, pages 243 – 248, Feb. 1997.