EINDHOVEN UNIVERSITY OF TECHNOLOGY

Analysis of an industrial system

Citation for published version (APA):
Kleijn, J. J. T., Reniers, M. A., & Rooda, J. E. (2003). Analysis of an industrial system. Formal Methods in System Design, 22(3), 249-282. DOI: 10.1023/A:1022901312673

DOI:
10.1023/A:1022901312673

Document status and date:
Published: 01/01/2003

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 01. Jul. 2019

Formal Methods in System Design, 22, 249–282, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

Analysis of an Industrial System

J.J.T. KLEIJN jeroen.kleijn@heineken.nl
Systems Engineering Group, Department of Mechanical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

M.A. RENIERS m.a.reniers@tue.nl
Technical Applications Group, Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

J.E. ROODA j.e.rooda@tue.nl
Systems Engineering Group, Department of Mechanical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Received August 17, 1999; Accepted April 5, 2001

Abstract. Studying industrial systems by simulation enables the designer to study their dynamic behaviour and to determine 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 μCRL. This enables us to give a correctness proof of the parameterised model and to study the model in isolation.

Keywords: industrial systems, formal methods, process algebra, verification

1. Introduction

Industry makes higher demands on methodologies used for the design of new factories and their machinery. 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. As a consequence, the same holds for the machinery inside. Building a new factory or improving a factory step by step is not an option anymore; the new factory should perform satisfactory from the beginning. Therefore, one must be able to predict system characteristics like throughput and cycle time and be able to rely on factory equipment; be able to assure certain behaviour. In addition, new factories must be realised within shorter periods. Products change faster, new products are developed faster and so must the systems needed to produce them.

Modelling is often used 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 [28]: physical (e.g., scale models), graphical

(e.g., engineering drawings), and symbolic (e.g., formal specifications). By far, most of the models used in computing science belong to the last category, while in other engineering disciplines also the other two are widely used.

In many applications, symbolic models are easier to construct and manipulate than physical or graphical ones. This is especially true for complex 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. Whether a language is suitable depends on the application at hand and hence languages come in different flavours.

Often modelling languages are subdivided into the following three categories: *continuous-time* languages, *discrete-event* languages, and their combination. Continuous-time modelling languages, such as ACSL [31] and Omola [2], are used for modelling physical systems where an infinite number of state changes can occur in a given finite time interval. Usually differential algebraic equations are used in such languages. Discrete-event modelling languages, such as SIMAN [33] and ExSpect [39], are used for describing systems that only allow a finite number of state changes in any finite time interval. Finally, languages exist that combine continuous-time and discrete-event features into one formalism. Examples are COSMOS [27], VHDL [10, 26, 35], and χ [1, 3, 12].

1.1. Simulating industrial systems with χ

Within the Systems Engineering group (http://se.wtb.tue.nl/), industrial systems (mostly production systems) are investigated in order to understand and improve their dynamic behaviour. They are modelled using a language especially designed for this purpose called χ [3, 38]. Validation of χ models can be done by means of simulation [1].

The formalism χ offers a language for specifying industrial systems including their control systems. Furthermore, the dynamic behaviour of these systems can be studied. As stated in the previous paragraph, it is possible to simulate χ models. 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 system characteristics like throughput and cycle time versus the work in process can be determined. 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. Money and time can be saved since changing a model costs only a fraction of making changes to the actual system.

The language χ has been tested and used extensively in many different industrial areas. Some of the projects that used χ and its simulator are mentioned below. Mostly, the objective was to reduce cycle time and to increase throughput by applying better scheduling strategies.

- *product oriented manufacturing*: the balancing of a car assembly line (Volvo/Mitsubishi),
- *process oriented manufacturing*: the design of an ASIC wafer fab (Philips [24, 36, 37]),
- *hybrid oriented production*: production optimisation in a beer brewery (Heineken) and the design of a fruit juice blending and packaging plant (Riedel [13]),

– *machinery*: design of control architectures and scheduling algorithms for medical equipment (TNO) and wafersteppers (ASML).

Although the simulation approach towards the validation of industrial systems turned out to be successful, the approach has 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, buffer sizes, 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, due to the fact that our validation is restricted to simulation we are forced to work with closed systems only. This means that the environment of the system must be modelled as well. As a consequence, we cannot study parts of the system in isolation. Thus, simulation does not allow a modular analysis of system behaviour. This makes analysis of complex industrial systems very difficult.

1.2. Verifying industrial systems with χ

The problems discussed in the previous paragraph lead to the insight that, apart from the ability to simulate, the ability to verify system properties is desired. A solution is to be found in proving system characteristics by using formal methods. Formal methods are the key to software verification in general, and in our case, the key to verifying χ programs. Firstly, it allows a thorough analysis of the possible behaviours of a specified system. 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. Formal methods also overcome the disadvantage last mentioned; it is possible to determine the behaviour of parts of the system in isolation, thus allowing a modular approach to the analysis of systems. Examples of such formal methods are the process algebras CSP [25], ACP [5], μCRL [18], and Petri Nets [34].

As a first step towards the verification of complex industrial systems specified in χ, we only consider the discrete-event part of χ in this article.

A formal method that can serve as a solution to the previously mentioned problems, for the discrete-event case, is the process algebra μCRL [18]. The description of continuous-time systems in μCRL is currently being investigated [23]. The process algebra μCRL is an extension of the process algebra ACP^τ [5] with a formal treatment of data. Data is present through so-called abstract data types. The language μCRL consists of only a few composition mechanisms that are expressive enough to describe large distributed systems. It has been the base for the development of a proof theory [17] and proof methodology [20] using which the verification of larger distributed systems in a precise logical way becomes a routine action.

The language μCRL and its proof methodology have been applied successfully to a number of case studies. These include, but are not limited to, the correctness proofs of a one bit sliding window protocol [7], a leader election protocol [14], a distributed summation

algorithm [21], and a bounded retransmission protocol [22]. The mathematical precision which characterises proofs in μCRL allows for the formalisation of these in proof checkers [6, 29].

In this paper, a case study is presented that deals with the earlier mentioned disadvantages of simulation. The case study considers a slightly adapted architecture of a sub-system of Philips' wafer fab MOS4YOU situated in Nijmegen, The Netherlands. We construct a χ model of this industrial system and map this model onto a μCRL 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* [20]. 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. This is in contrast with the χ simulator which requires the environment to be specified.

Furthermore, let us mention that we are aware of the fact that the studied model is relatively small, especially in comparison with the size of the χ models that are usually constructed. This is due to the fact that this case study is only our first attempt in applying formal techniques to the analysis of industrial systems modelled in the χ language.

2. The χ language

Within the χ language 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.

Each process is specified in an imperative way using guarded commands [11]. Consequently, χ exhibits non-determinism. Individual processes or groups of processes that operate concurrently communicate with each other by means of synchronous communication via channels. All channels are one-to-one connections between processes. This aspect of the language is based on CSP (Communicating Sequential Processes [25]).

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.

We only discuss a small part of the χ language, i.e. the language constructs needed for the case study discussed in Section 4. All other language features are omitted. A complete description of the χ language can be found in [1]. For now, the syntax as it is given in Table 1 suffices. The syntax is given in Backus-Naur Form (BNF) and the syntax of (boolean) expressions is omitted.

Assignments. The term $x := e$ denotes the assignment statement. It changes the value of x to e.

Table 1. Syntax of χ statements.

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

$PS ::=$	$x := e$	assignment statement
	$\mid PS_1;\ PS_2$	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
$GP ::=$	$b \to PS \mid GP \,[\!]\, GP$	
$GC ::=$	$b;\ ES \to PS \mid GC \,[\!]\, GC$	

Sequential composition. The term $PS_1;\ PS_2$ denotes the sequential composition of the statements PS_1 and PS_2, meaning that statement PS_2 is executed after the execution of statement PS_1 has finished.

Events. An event statement is either a send or a receive statement. The send statement $a \,!\, e$ can be used to send expression e via channel a to the other process connected with this channel. On the other hand, the receive statement $a \,?\, x$ can receive such an expression and assign it to variable x.

Selection. The selection statement can be used to denote that, depending on certain conditions (the guards), different statements should be executed. The construct $b \to PS$ is called a guarded command. A selection statement can combine one ore more guarded commands. Upon execution of a selection statement, all guards are evaluated. If none of the guards evaluates to true, execution of the selection statement deadlocks. In case more than one guard evaluates to true, one alternative is chosen non-deterministically and the corresponding statement is executed.

A repetitive selection statement can be used if we want a selection to be executed repeatedly as long as one or more guards evaluate to true. The repetition stops if all guards yield false, in that case the statement following the repetitive selection is executed. Note that therefore the repetitive selection cannot deadlock, as opposite to its non-repetitive variant.

Selective waiting. A selective waiting statement is somewhat like a selection statement with this difference that a possible communication is also part of the guard. Namely, besides the fact that all boolean expressions are evaluated to see which alternatives are valid, also opportunities to send or receive are checked. Only the alternatives that have a boolean expression that yields true and are able to actually communicate are considered to be valid. From these alternatives one alternative is chosen non-deterministically. As is the case for a selection statement; if none of the alternatives is valid, the statement deadlocks.

A repetitive selective waiting statement can be used if we want a selective waiting to be executed repeatedly as long as one or more boolean expressions evaluate to true and their

corresponding communications are possible. The repetition stops if all boolean expressions yield false or there are no opportunities to communicate. In that case, the statement following the repetitive selective waiting is executed. Note that also in this case the repetitive variant cannot deadlock.

3. The process algebra μCRL

In this section, we briefly introduce the process algebra μCRL and the proof methodology that is used for the case study.

3.1. *Specification of data in μCRL*

In μCRL [18] there is a simple, yet powerful mechanism to specify data. We use (unconditional equational) abstract data types with an explicit distinction between constructor functions and 'normal' functions [40]. The advantage of having such a simple language is that it can easily be explained and formally defined. Moreover, all properties of a data type must be explicitly denoted, and henceforth it is clear which assumptions can be used when proving properties about data or processes. A disadvantage is of course that even the simplest data types must be specified each time, and that there are no high level constructs that allow compact specification of complex data types.

Because booleans are used in the if-then-else construct in the language, the sort **Bool** must be declared in every μCRL specification. It is assumed that t (true) and f (false) are the only constructor functions of sort **Bool**. Furthermore, we assume that t and f are different and that every boolean expression equals one of them:

$$\neg(\mathsf{t} = \mathsf{f})$$
$$\neg(b = \mathsf{t}) \rightarrow b = \mathsf{f}$$

If in a specification t and f can be proven equal, for instance if the specification contains an equation $\mathsf{t} = \mathsf{f}$, then we say that the specification is inconsistent and it looses its meaning. Hence, for the booleans we declare

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

The division between constructors and mappings gives us general induction principles [40]. If a sort D is declared with a number of constructors, then we may assume that every term of sort D can be written as the application of a constructor to a number of arguments. So, if we want to prove a property $p(d)$ for all d of sort D, then we only need to provide proofs for $p(c_n(d_1, \ldots, d_n))$ for each n-ary constructor $c_n : S_1 \times \cdots \times S_n \rightarrow D$ and each d_i a term of sort S_i. If any of the arguments of c_n, say argument d_j, is of sort D then, as d_j is smaller than d, we may use that $p(d_j)$ as an induction hypothesis. If we apply this line of argumentation, we say we apply induction on D.

As an example of specifying data and reasoning about data, suppose that we have declared the natural numbers with constructors *zero* and *succ* and non-constructor function *plus* as follows:

sort N
func $zero :\to N$
$\quad succ : N \to N$
map $plus : N \times N \to N$
var $m, n : N$
rew $plus(m, zero) = m$
$\quad plus(m, succ(n)) = succ(plus(m, n))$

We can, for instance, derive that $plus(zero, n) = n$ for all n. We apply induction on N. First, we must show that $plus(zero, zero) = zero$, considering the case where $n = zero$. This is a trivial instance of the first axiom on addition. Second, we must show $plus(zero, succ(n')) = succ(n')$, assuming that n has the form $succ(n')$. In this case we may assume that the property to be proven holds already for n', i.e., $plus(zero, n') = n'$. Then we obtain $plus(zero, succ(n')) = succ(plus(zero, n')) = succ(n')$.

In the remainder of this paper we will omit all specifications of abstract data types.

3.2. Specification of processes in μCRL

The process algebra μCRL offers the following building blocks for describing processes: atomic actions, alternative and sequential composition, deadlock, process declarations, conditionals, alternative quantification, encapsulation, internal actions, hiding, and parallel composition. These will be explained below.

Atomic actions. Actions are abstract representations of events in the real world that is described. For instance sending the number 3 can be described by *send*(3) and boiling food can be described by *boil*(*food*) where 3 and *food* are terms declared by a data type specification. An action consists of an action name possibly followed by one or more data terms within brackets. The set of all action names that are declared in a μCRL specification is denoted by **Act**. In accordance with process algebras such a CCS, CSP and ACP, actions in μCRL are considered to be atomic.

Sequential and alternative composition. The two elementary operators to construct processes are the sequential composition operator $\cdot$ and the alternative composition operator $+$. The process $p \cdot q$ first performs the actions of p, until p terminates, and then continues with the actions in q. The process $p + q$ behaves like p or q, depending on which of the two performs the first action. Note that the sequential operator binds stronger than the alternative composition operator.

In Table 2 axioms A1–A5 are listed describing the elementary properties of the sequential and alternative composition operators. For instance, the axioms A1, A2 and A3 express that $+$ is commutative, associative and idempotent. In these and other axioms we use variables p, q and r that can be instantiated by process terms. In the sequel, we use the letter a for an

Table 2. Basic axioms for μCRL.

$p + q = q + p$	(A1)
$p + (q + r) = (p + q) + r$	(A2)
$p + p = p$	(A3)
$(p + q) \cdot r = p \cdot r + q \cdot r$	(A4)
$(p \cdot q) \cdot r = p \cdot (q \cdot r)$	(A5)

Table 3. Axioms for deadlock.

$p + \delta = p$	(A6)
$\delta \cdot p = \delta$	(A7)

action name, and we give each action a single argument $\bar{d}$ representing all its arguments. The letter c is used for any action or any of the special constants δ and τ, which are explained below.

Deadlock. The μCRL language contains a constant δ, expressing that no further actions can be performed. This constant is called *deadlock*. A typical property for δ is $p + \delta = p$; the choice in $p + q$ is determined by the first action performed by either p or q, and therefore one can never choose for δ. In other words, as long as there are alternatives deadlock is avoided. In Table 3, the axioms A6 and A7 characterise the main properties of δ.

Process declarations. Process declarations have the form:

$$X(x_1 : s_1, \ldots, x_n : s_n) = p.$$

Here X is the process name, x_i are variables, not clashing with the name of a constant function or a parameterless process or action name, and s_i are sort names. In this rule, process $X(x_1, \ldots, x_n)$ is declared to have the same (potential) behaviour as the process expression p. A process declaration must be considered as an equation in the ordinary mathematical sense. This means that with a declaration such as the one above an occurrence of $X(u_1, \ldots, u_n)$ can be replaced by $p(u_1/x_1, \ldots, u_n/x_n)$, or vice versa, $p(u_1/x_1, \ldots, u_n/x_n)$ may be replaced by $X(u_1, \ldots, u_n)$. The notation $p(e/v)$ represents the substitution of an expression e for every free occurrence of the variable v in process term p.

Conditional operator. The process expression $p \lhd b \rhd q$ where p and q are processes, and b is a data term of sort **Bool**, behaves like p if b is equal to t and if b is equal to f, behaves like q. This operator is called the conditional operator, and operates precisely as an *then_if_else* construct. Using the conditional operator data influences process behaviour. The conditional operator is characterised by axioms C1 and C2 in Table 4. There are no more axioms needed, because all required properties about the conditionals appear to be provable using the previously stated assumptions about the sort **Bool**.

Sum operator. The sum operator $\sum_{d:D} p$ behaves like the possibly infinite choice between $p(d_i/d)$ for any data term d_i taken from D, i.e. as $p(d_1/d) + p(d_2/d) + \cdots$. The sum

Table 4. Axioms for the conditional operator.

$p \lhd \mathsf{t} \rhd q = p$	(C1)
$p \lhd \mathsf{f} \rhd q = q$	(C2)

Table 5. Axioms for the sum operator.

$\sum_{d:D} p = p \quad \text{if } d \notin FV(p)$	(SUM1)
$\sum_{d:D} p = \sum_{d:D} p + p(e/d)$	(SUM3)
$\sum_{d:D}(p + q) = \sum_{d:D} p + \sum_{d:D} q$	(SUM4)
$(\sum_{d:D} p) \cdot q = \sum_{d:D}(p \cdot q) \quad \text{if } d \notin FV(q)$	(SUM5)

Table 6. Axioms for encapsulation.

$\partial_H(\delta) = \delta$	(DD)
$\partial_H(\tau) = \tau$	(DT)
$\partial_H(a(\bar{d})) = a(\bar{d}) \quad \text{if } a \notin H$	(D1)
$\partial_H(a(\bar{d})) = \delta \quad \text{if } a \in H$	(D2)
$\partial_H(p + q) = \partial_H(p) + \partial_H(q)$	(D3)
$\partial_H(p \cdot q) = \partial_H(p) \cdot \partial_H(q)$	(D)4
$\partial_H(\sum_{d:D} p) = \sum_{d:D} \partial_H(p)$	(SUM8)

operator $\sum_{d:D} p$ is a difficult operator, because it acts as a binder just like the lambda in the lambda calculus [4]. Conforming the λ-calculus, we allow α-conversion in the sum operator, and do not state this explicitly. Hence, we consider the expressions $\sum_{d:D} p$ and $\sum_{e:D} p(e/d)$ as equal. In Table 5 the axioms for the sum operator are listed. The notation $FV(p)$ represents the free variables of process p. We may not substitute the action $a(d)$ for p in the left hand side of SUM1 in Table 4, because this would cause d to become bound by the sum operator. This is expressed by d not in $FV(p)$. So, SUM1 is a concise way of saying that if d does not appear in p, then we may omit the sum operator in $\sum_{d:D} p$.

Encapsulation. Sometimes, we want to express that certain actions cannot happen, and must be blocked, i.e. renamed to δ. Generally, this is only done when we want to force this action into a communication. The encapsulation operator ∂_H ($H \subseteq$ **Act**) is especially designed for this task. In $\partial_H(p)$ it prevents all actions of which the action name is mentioned in H from happening. Typically, $\partial_{\{b\}}(a \cdot b(3) \cdot c) = a \cdot \delta$, where a, b and c are action names. The properties of ∂_H are given in Table 6.

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

The hidden action or internal action is denoted τ. It represents an action that can take place in a system, but cannot be observed directly. The internal action is meant for analysis

Table 7. Axioms for abstraction and internal actions.

$c \cdot \tau = c$	(B1)
$p \cdot (\tau \cdot (q + r) + q) = p \cdot (q + r)$	(B2)
$\tau_I(\delta) = \delta$	(TID)
$\tau_I(\tau) = \tau$	(TIT)
$\tau_I(a(\bar{d})) = a(\bar{d}) \quad \text{if } a \notin I$	(TI1)
$\tau_I(a(\bar{d})) = \tau \quad \text{if } a \in I$	(TI2)
$\tau_I(p + q) = \tau_I(p) + \tau_I(q)$	(TI3)
$\tau_I(p \cdot q) = \tau_I(p) \cdot \tau_I(q)$	(TI4)
$\tau_I(\sum_{d:D} p) = \sum_{d:D} \tau_I(p)$	(SUM9)

purposes, and hardly ever used in specifications, as it is very uncommon to specify that something unobservable must happen. A typical identity characterising the internal action τ is $a \cdot \tau \cdot p = a \cdot p$, with a an action and p a process term. It says that it is impossible to tell by observation whether or not internal actions happen after the a. Sometimes, the presence of internal actions can be observed, due to the context in which they appear. E.g. $a + \tau \cdot b \neq a + b$, as the left hand side can silently execute the τ, after which it only offers a b action, whereas the right hand side can always do an a. The difference between the two processes can be observed by insisting in both cases that the a happens. This is always successful in $a + b$, but may lead to a deadlock in $a + \tau \cdot b$. The axioms for internal actions, B1 and B2, are given in Table 7.

In order to make actions hidden, the hiding operator τ_I ($I \subseteq$ **Act**) is introduced, where I is a set of action names. The process $\tau_I(p)$ behaves as the process p, except that all actions with action names in I are renamed to τ. This is characterised by the axioms in Table 7.

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

It is possible to let processes p and q in $p \parallel q$ communicate. The possible communications are defined by a partial, commutative and associative communication function $\gamma : \mathbf{Act} \times \mathbf{Act} \to \mathbf{Act} \cup \{\delta, \tau\}$. Suppose that we have defined $\gamma(a, b) = \gamma(b, a) = c$. This means that if actions $a(\bar{d})$ and $b(\bar{d})$ can happen in parallel, they may synchronise and this synchronisation is denoted by $c(\bar{d})$. If two actions synchronise, their arguments must be exactly the same. In a communication definition it is required that action names a, b and c are declared with exactly the same data sorts. It is not necessary that these sorts are unique. It is for example perfectly right to declare the three actions both with a sort D and with a pair of sorts $D \times$ **Bool**.

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

Axioms that describe the parallel operator are in Table 8. In order to formulate the axioms two auxiliary parallel operators have been defined. The left merge $⫗$ is a binary operator

Table 8. Axioms for parallelism in μCRL.

Axiom	Name
$p \parallel q = p \mathbin{\|\!\!_} q + q \mathbin{\|\!\!_} p + p \mid q$	(CM1)
$c \mathbin{\|\!\!_} p = c \cdot p$	(CM2)
$c \cdot p \mathbin{\|\!\!_} q = c \cdot (p \parallel q)$	(CM3)
$(p + q) \mathbin{\|\!\!_} r = p \mathbin{\|\!\!_} r + q \mathbin{\|\!\!_} r$	(CM4)
$(\sum_{d:D} p) \mathbin{\|\!\!_} q = \sum_{d:D}(p \mathbin{\|\!\!_} q)$ if $d \notin FV(q)$	(SUM6)
$a(\bar{d}) \mid a'(\bar{e}) = \begin{cases} \gamma(a, a')(\bar{d}) \lhd eq(\bar{d}, \bar{e}) \rhd \delta & \text{if } \gamma(a, a') \text{ defined} \\ \delta & \text{otherwise} \end{cases}$	(CF)
$\delta \mid c = \delta$	(CD1)
$c \mid \delta = \delta$	(CD2)
$\tau \mid c = \delta$	(CT1)
$c \mid \tau = \delta$	(CT2)
$c \cdot p \mid c' = (c \mid c') \cdot p$	(CM5)
$c \mid c' \cdot p = (c \mid c') \cdot p$	(CM6)
$c \cdot p \mid c' \cdot q = (c \mid c') \cdot (p \parallel q)$	(CM7)
$(p + q) \mid r = p \mid r + q \mid r$	(CM8)
$p \mid (q + r) = p \mid q + p \mid r$	(CM9)
$(\sum_{d:D} p) \mid q = \sum_{d:D}(p \mid q)$ if $d \notin FV(q)$	(SUM7)
$p \mid (\sum_{d:D} q) = \sum_{d:D}(p \mid q)$ if $d \notin FV(p)$	(SUM7′)

that behaves exactly as the parallel operator, except that its first action must come from the left hand side. The communication merge | is also a binary operator behaving as the parallel operator, except that the first action must be a synchronisation between its left and right operand. The core law for the parallel operator is CM1 in Table 8. It says that in $p \parallel q$ either p performs the first step, represented by the summand $p \mathbin{\|\!\!_} q$, or q can do the first step, represented by $q \mathbin{\|\!\!_} p$, or the first step of $p \parallel q$ is a communication between p and q, represented by $p \mid q$. All other axioms in Table 8 are designed to eliminate the parallel operators in favour of the alternative and the sequential operator.

The axioms provided for the μCRL language describe which processes are considered equivalent. For μCRL a structured operational semantics can be found in [18]. The equivalence described by the axioms corresponds with the notion of rooted branching bisimulation on transition systems.

3.3. *Proof methodology*

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 send and receive actions. First, the send and receive actions that did not result in a communication are encapsulated. Then, the internal actions (i.e., the actions within the system itself and without a direct link to the environment), are abstracted from. Given a specification *Spec* describing the external behaviour that the implementation should satisfy, it should be the case that the implementation (after the previously mentioned encapsulation and abstraction) equals this

specification:

$$Spec = \tau_I(\partial_H(p_1 \parallel p_2 \parallel \cdots \parallel p_n)).$$

To prove such an equivalence, we use the notion of focus points and convergent process operators [20]. It forms a strategy for finding algebraic correctness proofs for communication systems described in μCRL.

Proving that two processes are equivalent 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 (i.e. there is no infinite sequence of internal steps), 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 infinite sequence of consecutive τ-actions 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 *Clustered Linear Process Equations* (*C-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. The C-LPE format resembles I/O-automata [30] and Unity processes [9].

Definition 1 (*C-LPE*). Let $Act \subseteq \mathbf{Act} \cup \{\tau\}$ be a finite set of actions and let D be an arbitrary data sort. A Clustered Linear Process Equation (*C-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 $b_a : D \to E_a \to \mathbf{Bool}$.

The *C-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 *C-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 2 (Focus condition). The focus condition $FC_p(d)$ of *C-LPE* p in state d is the formula

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

Definition 3 (Matching criteria). Let $p(d{:}D)$ and $q(d'{:}D')$ be *C-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 $e_a{:}E_a$

$$p \text{ is convergent,} \tag{1}$$

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

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

$$FC_p(d) \wedge b'_a(h(d), e_a) \to b_a(d, e_a), \tag{4}$$

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

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

In [20], 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 1. *Let $p(d{:}D)$ and $q(d'{:}D')$ be C-LPEs and $h : D \to D'$ a state mapping. If q does not contain τ-steps, then p and q are equivalent*: $p = q$.

4. The case study

This particular case study deals with an industrial 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

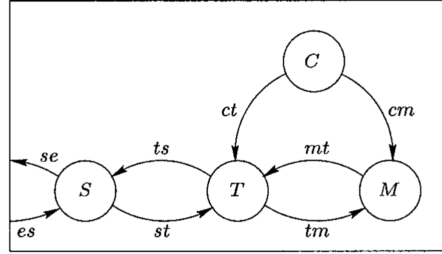

Figure 1. Industrial system.

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.

Firstly, we specify the χ models of the specification (Ψ) and implementation of the industrial system. From these χ models we derive the *C-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 $\Xi = \Psi$.

Products are represented by naturals. Batches are lists (notation: *type**) of products. In χ the types 'prod' and 'batch' are then declared by

$$\begin{aligned} \mathsf{type}\ &\text{prod} = \text{nat}, \\ &\text{batch} = \text{prod}^*. \end{aligned}$$

Before we present the χ models, the predefined functions *take* and *drop* are given. The natural numbers are denoted by N. 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. The definitions of the functions $+\!\!+$, *len*, *hd* and *tl* are omitted.

Definition 4. Let T be an arbitrary data type. Then for all $x \in T$, $xs \in T^*$ and $n \in N$, the functions $take, drop : T^* \times N \to T^*$ 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 intuition behind the two functions defined above can be expressed by $q = take(q, n) +\!\!+ drop(q, n)$ for all q and n.

The desired external behaviour, to be specified in process Ψ, can be described as follows: products enter the 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. We can abbreviate an expression '*true*; *ES*' to *ES*. This is done in the process descriptions of Ψ and S.

```
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 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 STMC(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 *C-LPE* models for the specification and the abstract implementation. We shortly discuss the way to obtain a *C-LPE* model from a corresponding χ model and illustrate that approach for the specification.

The main issue when translating a χ model into its μCRL equivalent is that one has to take into account a second programming paradigm. Within the χ language the process part and the arithmetic part are mixed and specified in an imperative way. That is, explicit sequences of steps are specified in order to realise a certain result. The μCRL language, on the other hand, has a clear distinction between the process part and the arithmetic part and it uses a combined approach. The process part is imperative, supporting recursion, while the arithmetic part is declarative. The latter implies that changes of variables are described in terms of functions. The state parameters are included in the process' parameter list. Arithmetic is handled by updating those state parameters when a process is recursively called.

Table 1 shows us that, in essence, χ processes are built from the assignment statement ($x := e$), the send statement ($a\,!\,e$), the receive statement ($a\,?\,x$), and boolean expressions which act as guards (b). A *C-LPE*, as defined in Definition 1, is built from actions, recursive calls of that particular process and guards. We represent a send and receive statement by a send and receive action, $s_a(e)$ and $r_a(x)$, and in addition, we also define a communication action $c_a(e)$. Such a communication action results from a successful synchronised execution of a send and receive action according to a predefined communication function γ: $\gamma(s_a, r_a) = c_a$. In stead of the assignment statement we write e/v if a variable v is replaced by an expression e.

Let us now, as an example, consider the specification. Even though χ does not have an explicit notion of states, it is not hard to see that process Ψ is described by list xs and parameter s. Furthermore, 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 μCRL processes Ψ and S. They represent the next product to be received via channel es. The μCRL process Ψ is then parameterised by x_Ψ, xs_Ψ, and s. As a result, the state space is then given by $D_\Psi = N \times N^* \times N^{>0}$. Note that, in addition to updating list xs_Ψ after we received a new product,

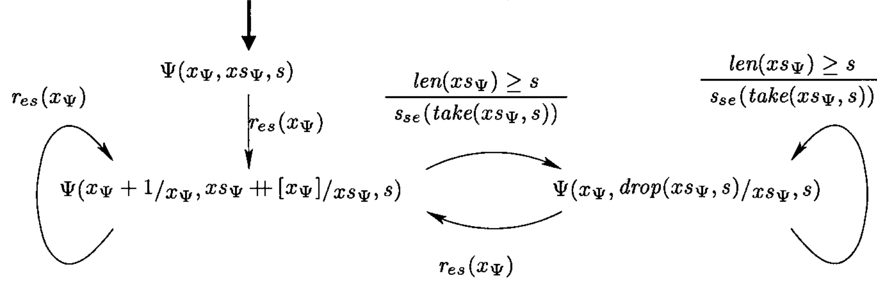

Figure 2. Transition diagram of process Ψ.

we now also need to increase x_Ψ by 1. Also note that, in order to increase readability of the expressions to follow, we only show those parameters of a process in its parameter list that are actually modified.

This leads to the following definition of the specification in *C-LPE* format.

Definition 5. The specification is given by the process $\Psi(1, [], s)$, where

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

The transition diagram of process Ψ is depicted in figure 2 below.

Before we construct the μCRL model for the abstract implementation, we first specify its components. The components are the processes S, T, M and C. Because the *C-LPE* format forces us to maintain former sequential compositions within a process by means of some extra arithmetic, extra state parameters j_t, j_m, $j_c \in N$ are introduced in the processes T, M and C. These parameters are used in guards to enforce the execution of a specific action as the successor of a certain previous action.

Definition 6. For $(x_s, xs_s, ys_s) \in D_S$, where $D_S = N \times N^* \times (N^*)^*$, we define

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

For $(xs_t, z_t, j_t) \in D_T$, where $D_T = N^* \times \mathbf{Bool} \times N$, we define

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

$$+ s_{ts}(xs_t) \cdot T([]/xs_t, 2/j_t) \triangleleft \neg z_t \wedge j_t = 1 \triangleright \delta$$
$$+ \sum_{z:\mathbf{Bool}} r_{ct}(z) \cdot T(z/z_t, 0/j_t) \triangleleft j_t = 2 \triangleright \delta.$$

For $(xs_m, z_m, j_m) \in D_M$, where $D_M = N^* \times \mathbf{Bool} \times N$, we define

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

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

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

These processes communicate over the channels as depicted in figure 1. For this case study, γ is defined by

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

for all $ch \in \{es, st, tm, mt, ts, se, ct, cm\}$.

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

$$H = \{s_{ch}, r_{ch} \mid ch \in \{st, tm, mt, ts, ct, cm\}\}.$$

The implementation can then be described by

$$\partial_H(S(1, [], []) \parallel T([], \mathrm{t}, 0) \parallel M([], \mathrm{t}, 0) \parallel C(0, 0, s)).$$

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

$$I = \{c_{ch} \mid ch \in \{st, tm, mt, ts, ct, cm\}\}.$$

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

$$\tau_I(\partial_H(S(1, [], []) \parallel T([], \mathrm{t}, 0) \parallel M([], \mathrm{t}, 0) \parallel C(0, 0, s))).$$

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 numbered for referring purposes.

Theorem 2. *The abstract implementation Ξ is the process*

$$\Xi(1, [], [], [], \mathrm{t}, 0, [], \mathrm{t}, 0, 0, 0, s),$$

where

$$
\begin{aligned}
&\Xi(x_s, xs_s, ys_s, xs_t, z_t, j_t, xs_m, z_m, j_m, i_c, j_c, s)\\
&\quad = r_{es}(x_s) \cdot \Xi(x_s + 1/x_s, xs_s +\!\!+ [x_s]/xs_s)\\
&\qquad + \tau_1 \cdot \Xi(tl(xs_s)/xs_s, xs_t +\!\!+ [hd(xs_s)]/xs_t, 1/j_t)\\
&\qquad\qquad \lhd\, len(xs_s) > 0 \wedge z_t \wedge j_t = 0 \rhd \delta\\
&\qquad + \tau_2 \cdot \Xi([]/xs_t, 2/j_t, xs_m +\!\!+ [hd(xs_t)]/xs_m, 1/j_m)\\
&\qquad\qquad \lhd\, z_t \wedge j_t = 1 \wedge z_m \wedge j_m = 0 \rhd \delta\\
&\qquad + \tau_3 \cdot \Xi(xs_t +\!\!+ xs_m/xs_t, 1/j_t, []/xs_m, 1/j_m)\\
&\qquad\qquad \lhd\, \neg z_t \wedge j_t = 0 \wedge \neg z_m \wedge j_m = 0 \rhd \delta\\
&\qquad + \tau_4 \cdot \Xi(ys_s +\!\!+ [xs_t]/ys_s, []/xs_t, 2/]j_t)\\
&\qquad\qquad \lhd\, \neg\, z_t \wedge j_t = 1 \rhd \delta\\
&\qquad + s_{se}(hd(ys_s)) \cdot \Xi(tl(ys_s)/ys_s)\\
&\qquad\qquad \lhd\, len(ys_s) > 0 \rhd \delta\\
&\qquad + \tau_5 \cdot \Xi(i_c < s/z_t, 0/j_t, 1/j_c)\\
&\qquad\qquad \lhd\, j_t = 2 \wedge j_c = 0 \rhd \delta\\
&\qquad + \tau_6 \cdot \Xi(i_c < s/z_m, 0/j_m, (i_c + 1) \bmod (s + 1)/i_c, 0/j_c)\\
&\qquad\qquad \lhd\, j_m = 1 \wedge j_c = 1 \rhd \delta.
\end{aligned}
$$

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 industrial 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 $D_{\Xi'} = N \times N$, 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 2).

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, [], [], [], t, 0, [], t, 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 7. Let T be an arbitrary data type. Then for all $xs : T^*, xss : (T^*)^*$ the function $flat : (T^*)^* \to T^*$ is defined by

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

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

$$list(d) = \begin{cases} flat(ys_s) \mathbin{+\!\!+} (xs_t \mathbin{+\!\!+} xs_m) \mathbin{+\!\!+} xs_s \mathbin{+\!\!+} [x_s] & \text{if } \neg z_t, \\ flat(ys_s) \mathbin{+\!\!+} (xs_m \mathbin{+\!\!+} xs_t) \mathbin{+\!\!+} xs_s \mathbin{+\!\!+} [x_s] & \text{if } z_t. \end{cases}$$

The functions $min, max : D_\Xi \to N$ are defined by

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

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

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

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.

Table 9. Invariants.

$(\neg z_t \wedge j_t = 1) \vee j_t = 2 \rightarrow j_c = 0$	(7)
$\neg z_t \wedge j_t = 1 \rightarrow len(xs_t) = s$	(8)
$\neg j_t = 1 \rightarrow xs_t = []$	(9)
$j_m = 0 \rightarrow j_t = 0 \vee (z_t \wedge j_t = 1)$	(10)
$\neg z_m \leftrightarrow i_c = s$	(11)
$j_m = 0 \rightarrow (z_t \leftrightarrow z_m)$	(12)
$j_m = 0 \rightarrow len(xs_m) = i_c$	(13)
$(j_t = 0 \vee (z_t \wedge j_t = 1)) \wedge j_m = 1 \rightarrow j_c = 1$	(14)

Definition 9. The functions $h_1 : D_\Xi \rightarrow D_{\Xi'}$, $h_2 : D_{\Xi'} \rightarrow D_\Psi$, and $h : D_\Xi \rightarrow D_\Psi$ are defined by

$$h_1(d) = (min(d), max(d)),$$
$$h_2(d) = (np(d'), pp(d')),$$
$$h(d) = h_2(h_1(d)).$$

Before we prove that the matching criteria hold for our state mapping h, let us first prove some general properties of h and Ξ. For instance, we need invariants which describe how the system parameters of the abstract implementation relate to each other. The ones required in the sequel are given in Table 9. Since one can easily verify that these invariants hold for the abstract implementation, proofs are omitted. One of the disadvantages of the approach towards verification in this paper is that the invariants used do not correspond well with intuition. This is due to the fact that we had to introduce additional state variables while linearising the system. Such state variables usually play a prominent rôle in invariants. This makes it hard to give a good intuition for such invariants.

Consider the function $inc : N^* \rightarrow \mathbf{Bool}$. This function checks if a list of numbers $xs \in N^*$ is increasing with increments of value 1. If this is the case, the function inc yields true, else it yields false.

Definition 10. The function $inc : N^* \rightarrow \mathbf{Bool}$ is, for all $x, x_1, x_2 \in N$ and $xs \in N^*$, defined by

$$inc([]) = \mathsf{t},$$
$$inc([x]) = \mathsf{t},$$
$$inc([x_1] +\!\!+ [x_2] +\!\!+ xs) = inc([x_2] +\!\!+ xs) \wedge x_2 = x_1 + 1.$$

Lemma 1. *Without any further proof, we claim that the following properties hold for the function inc, with $s, x \in N$, $xs \in N^*$ and '$toe(xs)$' last element of xs*

1. $xs \neq [] \rightarrow inc(xs) \wedge x = toe(xs) + 1 \leftrightarrow inc(xs +\!\!+ [x])$,
2. $inc(xs) \rightarrow inc(drop(xs, s))$.

Proposition 1. $inc(pp(m, n))$.

Proof: Due to Definition 8 all lists $[m, \ldots, n]$ are increasing. □

Proposition 2. $\Xi(d_0) \vdash max(d) = x_s - 1$.

Proof: For all reachable states of our abstract implementation Ξ (Theorem 2), 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 $\Xi(d_0) \vdash max(d) = x_s - 1$. □

Proposition 3. $\forall_{\tau \in \{\tau_1,\ldots,\tau_6\}}(\Xi(d_0) \vdash b_\tau(d) \rightarrow list(d) = list(d^\star))$.

Proof: Proposition 3 holds for τ_1 up to τ_4 since for all lists xs_0, xs_1 and xs_2, we have that $(xs_0 +\!\!+ xs_1) +\!\!+ xs_2 = xs_0 +\!\!+ (xs_1 +\!\!+ xs_2)$ (associativity). Considering τ_5, Inv. (9) holds. Therefore, whether z_t equals true or false has no influence on the result of the function *list* defined in Definition 7. We always have $xs_t +\!\!+ xs_m = xs_m +\!\!+ xs_t$ and thus Proposition 3 also holds for the action τ_5. Proposition 3 trivially holds for the action τ_6 since none of the relevant parameters of the function *list* are modified. □

Proposition 4. $\Xi(d_0) \vdash inc(list(d))$.

Proof: The proposition is proved by induction. We have $\Xi(d_0) \vdash inc(list(d_0))$ since $inc(list(d_0)) = inc([1])$, which yields true according to Definition 10. Let us furthermore prove

$$\forall_{a \in Act}(\Xi(d_0) \vdash b_a(d) \wedge inc(list(d)) \rightarrow inc(list(d^\star))).$$

– Action $r_{es}(x_\Xi)$: d : $inc(flat(ys_s) +\!\!+ \cdots +\!\!+ xs_s +\!\!+ [x_\Xi])$
  $d^\star$: $inc(flat(ys_s) +\!\!+ \cdots +\!\!+ (xs_s +\!\!+ [x_\Xi]) +\!\!+ [x_\Xi + 1])$
  According to Property 1 of Lemma 1, we have that

$$\Xi(d_0) \vdash b_{r_{es}(x_\Xi)}(d) \wedge inc(list(d)) \rightarrow inc(list(d^\star)).$$

– Action $s_{se}(hd(ys_s))$: d : $inc(flat(ys_s) +\!\!+ \cdots +\!\!+ [x_\Xi])$,
  $d^\star$: $inc(flat(tl(ys_s)) +\!\!+ \cdots +\!\!+ [x_\Xi])$.
  Within the abstract implementation Ξ it is reassured that we will not have $tl(ys_s) = tl([])$ by the condition $len(ys_s) > 0$. Therefore, according to Property 2 of Lemma 1, we have that

$$\Xi(d_0) \vdash b_{s_{se}(hd(ys_s))}(d) \wedge inc(list(d)) \rightarrow inc(list(d^\star)).$$

– Actions τ_1 up to τ_6: From Proposition 3, it follows that

$$\forall_{\tau \in \{\tau_1,\ldots,\tau_6\}}, (\Xi(d_0) \vdash b_\tau(d) \rightarrow list(d) = list(d^\star)).$$

  and thus we have that

$$\forall_{\tau \in \{\tau_1,\ldots,\tau_6\}}(\Xi(d_0) \vdash b_\tau(d) \wedge inc(list(d)) \rightarrow inc(list(d^\star))).$$ □

Proposition 5. $\Xi(d_0) \vdash min(d) > max(d) \rightarrow min(d) = x_\Xi$.

Proof: Suppose that $\Xi(d_0) \vdash min(d) > max(d)$. From Proposition 2, it follows that $\Xi(d_0) \vdash max(d) = x_\Xi - 1$ and as a consequence, we have that $\Xi(d_0) \vdash min(d) \geq x_\Xi$. From Definition 7 and Proposition 4 it follows that that $\Xi(d_0) \vdash min(d) \leq x_\Xi$. Thus, $\Xi(d_0) \vdash min(d) > max(d) \rightarrow min(d) = x_\Xi$. □

Next, we show that the state mapping h satisfies the matching criteria, which implies that specification and abstract implementation of our system are equivalent (Theorem 1). This is done by discussing the criteria one by one and proving that each criterion holds.

Criterion 1. Ξ *is convergent*.

Proof: As our implementation Ξ does not contain any τ loops, we have that in a cone of the implementation Ξ every internal action τ constitutes progress towards a focus point. Consequently, Ξ is convergent. □

Criterion 2. $\forall_{e_\tau : E_\tau}(b_\tau(d, e_\tau) \rightarrow h(d) = h(g_\tau(d, e_\tau)))$.

Figure 3 gives a graphical representation of Criterion 2. The transition from one state to another is represented by a vertical arrow accompanied by a possible condition and its action. State mappings between the implementation and the specification are represented by dashed arrows.

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.

Proof: Since $h(d) = h_2(h_1(d))$, we can write Criterion 2 as $b_\tau(d) \rightarrow h_2(h_1(d)) = h_2(h_1(g_\tau(d)))$. Then it suffices to prove that $b_\tau(d) \rightarrow h_1(d) = h_1(g_\tau(d))$. Together with $h_1(d) = (min(d), max(d))$ (Definition 9) and by using $d^\star = g_\tau(d)$ as a shorthand notation, this comes to proving the following two propositions

$$b_\tau(d) \rightarrow min(d) = min(d^\star), \quad (15.1)$$

$$b_\tau(d) \rightarrow max(d) = max(d^\star). \quad (15.2)$$

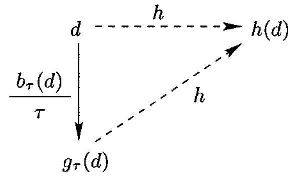

Figure 3. Criterion 2.

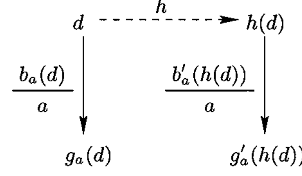

Figure 4. Criterion 3.

Let us prove (15.1). Suppose that $\forall_{\tau \in \{\tau_1,\ldots,\tau_6\}}(\Xi(d_0) \vdash b_\tau(d))$. From Proposition 3 it follows that $\Xi(d_0) \vdash list(d) = list(d^\star)$. So, $\Xi(d_0) \vdash min(d) = min(d^\star)$.

Let us furthermore prove (15.2). It trivially holds since from Proposition 2 it follows that $\Xi(d_0) \vdash max(d) = x_\Xi - 1$ and the parameter x_Ξ is not modified after any of the τ-steps τ_1 up to τ_6. □

Criterion 3. $\forall_{a \in Act\setminus\{\tau\}}\forall_{e_a:E_a}(b_a(d, e_a) \to b'_a(h(d), e_a))$.

Figure 4 gives a graphical representation of Criterion 3. It 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 system, we have that both specification and abstract implementation can always receive a new product. Furthermore, we can prove that when $len(ys_s) > 0$ we also have $len(xs_\Psi) \geq s$. This means that when the abstract implementation is able to deliver a completed batch, the specification is able to do the same.

Proof: Proving that the state mapping h matches the third criterion means proving that it holds for $a \in \{r_{es}(x_\Xi), s_{se}(hd(ys_s))\}$. It is easy to see that this criterion is satisfied in case of $r_{es}(x_\Xi)$ because in that case we have $b'_a(h(d)) = \mathsf{t}$. In case of $s_{se}(hd(ys_s))$, we have that $b_a(d)$ equals $len(ys_s) > 0$ and $b'_a(h(d))$ equals $len(pp(min(d), max(d))) \geq s$. Then it remains to prove that

$$len(ys_s) > 0 \to len(pp(min(d), max(d))) > s.$$

Suppose $len(ys_s) > 0$, then at least a number of products equal to the batch size is present in the system. From Proposition 4 it follows that $\Xi(d_0) \vdash inc(list(d))$. As a result, $max(d) \geq min(d) + s - 1$ and due to the definition of function pp, we have $len(pp(min(d), max(d))) \geq s$. □

Criterion 4. $\forall_{a \in Act\setminus\{\tau\}}\forall_{e_a:E_a}(FC_\Xi(d) \wedge b'_a(h(d), e_a) \to b_a(d, e_a))$.

Figure 5 gives a graphical representation of Criterion 4. It 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 delivery of processed batches, we can prove that when $len(xs_\Psi) \geq s$, we can also realise $len(ys_s) > 0$, the possibility to deliver a completed batch in the abstract implementation, provided that no τ step is enabled.

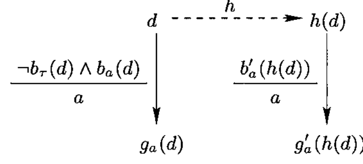

Figure 5. Criterion 4.

Proof: Proving that the state mapping h matches the fourth criterion means proving that it holds for $a \in \{r_{es}(x_\Xi), s_{se}(hd(ys_s))\}$. It is easy to see that this criterion is satisfied in case of action $r_{es}(x_\Xi)$ as $b_a(d) = \mathsf{t}$.

In case of $s_{se}(hd(ys_s))$, $b'_a(h(d)) = len(pp(min(d), max(d))) \geq s$ and $b_a(d) = len(ys_s) > 0$. The focus condition equals $\neg b_\tau(d)$ which yields $\bigwedge_{1 \leq i \leq 6} \neg b_{\tau_i}(d)$. The proposition to prove then is as follows

$$\neg b_\tau(d) \wedge len(pp(min(d), \max(d))) \geq s \rightarrow len(ys_s) > 0. \tag{16}$$

Let us prove this by means of case distinction based on the lists ys_s, xs_t, xs_m and xs_s.

Case $len(ys_s) > 0$. The conclusion of (16) yields t and we are done.

Case $len(xs_t) > 0$. We are only interested in the case $j_t = 1$ since $\neg j_t = 1 \rightarrow xs_t = []$ (Inv. (9)).

- Suppose $z_t \wedge j_t = 1$.
  - Suppose $j_m = 0$. According to Inv. (12), $j_m = 0 \rightarrow (z_t \leftrightarrow z_m)$, we have $z_m \wedge j_m = 0$. This implies that τ_2 is enabled and the premise yields f.
  - Suppose $j_m = 1$. According to Inv. (14), $(j_t = 0 \vee (z_t \wedge j_t = 1)) \wedge j_m = 1 \rightarrow j_c = 1$, we have $j_m = 1 \wedge j_c = 1$. This implies that τ_6 is enabled and the premise yields f.
- Suppose $\neg z_t \wedge j_t = 1$. This implies that τ_4 is enabled and the premise yields f.

Case $len(xs_m) > 0$.

- Suppose $j_m = 0$. In that case, Inv. (10), $j_m = 0 \rightarrow j_t = 0 \vee (z_t \wedge j_t = 1)$ and Inv. (12), $j_m = 0 \rightarrow (z_t \leftrightarrow z_m)$, apply.
  - Suppose $z_t \wedge j_t = 0$.
    * Suppose $len(xs_m) < s$. In case $len(pp(min(d), max(d))) \geq s$, then we have that $max(d) \geq min(d) + s - 1$ according to Definition 10. Together with the fact that $len(xs_m) < s$, $len(ys_s) = 0$ and $len(xs_t) = 0$, this implies that $len(xs_s) > 0$. Consequently, τ_1 is enabled and the premise yields f. In case $len(pp(min(d), max(d))) < s$ the premise also yields f.
    * Suppose $len(xs_m) = s$. This case cannot occur here since according to Inv. (13), $j_m = 0 \rightarrow len(xs_m) = i_c$, we should have $i_c = s$, which implies that we also

have $\neg z_m$ (Inv. (11), $\neg z_m \leftrightarrow i_c = s$). This is in contradiction with Inv. (12), $j_m = 0 \to (z_t \leftrightarrow z_m)$.

- Suppose $\neg z_t \wedge j_t = 0$. This implies that τ_3 is enabled and the premise yields f.
- Suppose $z_t \wedge j_t = 1$. This implies that τ_2 is enabled and the premise yields f.

– Suppose $j_m = 1$.

- Suppose $j_t = 0 \vee (z_t \wedge j_t = 1)$. In that case, we have $j_m = 1 \wedge j_c = 1$ according to Inv. (14), $(j_t = 0 \vee (z_t \wedge j_t = 1)) \wedge j_m = 1 \to j_c = 1$. This implies that τ_6 is enabled and the premise yields f.
- Suppose $\neg z_t \wedge j_t = 1$. This implies that τ_4 is enabled and the premise yields f.
- Suppose $j_t = 2$. In that case we have $j_t = 2 \wedge j_c = 0$ according to Inv. (7), $(\neg z_t \wedge j_t = 1) \vee j_t = 2 \to j_c = 0$. This implies that τ_5 is enabled and the premise yields f.

Case $len(xs_s) > 0$.

– Suppose $z_t \wedge j_t = 0$. This implies that τ_1 is enabled and the premise yields f.
– Suppose $\neg z_t \wedge j_t = 0$.

- Suppose $j_m = 0$. According to Inv. (12), $j_m = 0 \to (z_t \leftrightarrow z_m)$, τ_3 is enabled and the premise yields f.
- Suppose $j_m = 1$. According to Inv. (14), $(j_t = 0 \vee (z_t \wedge j_t = 1)) \wedge j_m = 1 \to j_c = 1$, τ_6 is enabled and the premise yields f.

– Suppose $z_t \wedge j_t = 1$. In case $j_m = 0$, according to Inv. (12), $j_m = 0 \to (z_t \leftrightarrow z_m)$, we have that τ_2 is enabled and the premise yields f. In case $j_m = 1$, according to Inv. (14), $(j_t = 0 \vee (z_t \wedge j_t = 1)) \wedge j_m = 1 \to j_c = 1$, we have that τ_6 is enabled and the premise yields f.
– Suppose $\neg z_t \wedge j_t = 1$. This implies that τ_4 is enabled and the premise yields f.
– Suppose $j_t = 2$. According to Inv. (7), $(\neg z_t \wedge j_t = 1) \vee j_t = 2 \to j_c = 0$, we have that τ_5 is enabled and the premise yields f.

Case $len(xs_s) = 0$. In this case $min(d) = x_\Xi$, and $max(d) = x_\Xi - 1$. Then the premise yields f because $len(pp(x_\Xi, x_\Xi - 1)) = 0$. □

Criterion 5. $\forall_{a \in Act \setminus \{\tau\}} \forall_{e_a : E_a} (b_a(d, e_a) \to f_a(d, e_a) = f'_a(h(d), e_a))$.

Figure 6 gives a graphical representation of Criterion 5. It says that corresponding external actions carry the same data parameter (modulo h).

Proof: Proving that the state mapping h matches the fifth criterion means proving that it holds for $a \in \{r_{es}(x_\Xi), s_{se}(hd(ys_s))\}$. That gives us the following propositions for the actions $r_{es}(x_\Xi)$ and $s_{se}(hd(ys_s))$ respectively

$$\mathsf{t} \to x_\Xi = x_\Psi, \tag{17.1}$$

$$len(ys_s) > 0 \to hd(ys_s) = take(xs_\Psi, s). \tag{17.2}$$

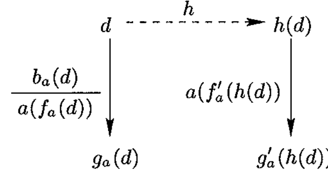

Figure 6. Criterion 5.

For the proof of (17.1), consider the following computation

$$\begin{aligned} x_\Psi &= np(min(d_\Xi), max(d_\Xi)) \\ &= max(d_\Xi) + 1 \\ &= (x_\Xi - 1) + 1 \\ &= x_\Xi. \end{aligned}$$

For the proof of (17.2), consider the following

$$\begin{aligned} take(xs_\Psi, s) &= take(pp(min(d_\Xi), max(d_\Xi)), s) \\ &= take([min(d_\Xi), \ldots, max(d_\Xi)], s). \end{aligned}$$

Then it remains to prove that

$$len(ys_s) > 0 \to hd(ys_s) = take([min(d_\Xi), \ldots, max(d_\Xi)], s).$$

This holds if the following propositions hold

$$len(ys_s) > 0 \to inc(hd(ys_s)) \wedge inc(take([min(d_\Xi), \ldots, max(d_\Xi)], s)), \quad (18.1)$$
$$len(ys_s) > 0 \to len(hd(ys_s)) = len(take([min(d_\Xi), \ldots, max(d_\Xi)], s)), \quad (18.2)$$
$$len(ys_s) > 0 \to hd(hd(ys_s)) = hd(take([min(d_\Xi), \ldots, max(d_\Xi)], s)). \quad (18.3)$$

For the proof of (18.1), consider the following. From Proposition 4, it follows that $\Xi(d_0) \vdash inc(list(d_\Xi))$. In that case we also have $inc(hd(ys_s))$ because of the definition of the functions *list* and *flat* (Definition 7). Furthermore, due to the definitions of functions *take* and *inc*, Definitions 4 and 10,

$$\forall_{xs \in N^*} \forall_{s \in N} (inc(xs) \to inc(take(xs, s)).$$

So, we also have $inc(take([min(d_\Xi), \ldots, max(d_\Xi)], s))$ and thus (18.1) holds.

For the proof of (18.2), consider the following. For the case that $len(ys_s) > 0$, let us denote ys_s as $[y_0, \ldots, y_n]$ with $y_i \in N^*$. Now we can prove that $len(y_i) = s$ for each i such that $0 \leq i \leq n$. Looking at Theorem 2 and using the Inv. (8), (11) and (13), we can see that this holds. Thus, $len(ys_s) > 0 \to len(hd(ys_s)) = s$ and from the definition of the function *take*, Definition 4, and Definition 5 it follows that also $len(xs_\Psi) \geq s \to len(take(xs_\Psi, s)) = s$. So, both $hd(ys_s)$ and $take([min(d_\Xi), \ldots, max(d_\Xi)], s)$ have length s and thus (18.2) holds.

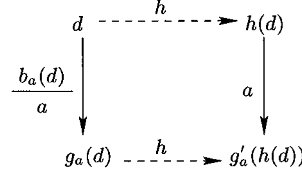

Figure 7. Criterion 6.

For the proof of (18.3), consider the following. From Definition 5 it follows that $len(xs_\Psi) \geq s$. In that case, consider the following computation

$$\begin{aligned} hd(take([min(d_\Xi), \ldots, max(d_\Xi)], s)) &= min(d_\Xi) \\ &= hd(list(d_\Xi)) \\ &= hd(flat(ys_s)) \\ &= hd(hd(ys_s)). \end{aligned}$$

□

Criterion 6. $\forall_{a \in Act \setminus \{\tau\}} \forall_{e_a : E_a} (b_a(d, e_a) \to h(g_a(d, e_a)) = g'_a(h(d), e_a))$.

Figure 7 gives a graphical representation of Criterion 6. It 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 $b_a(d) = \mathsf{t}$ and we have that $g_a(d) = d[x_s + 1/x_s, xs_s \mathbin{+\!\!+} [x_s]/xs_s]$. The latter is denoted as $d^\star$ in the sequel. We have

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

Then, the following propositions remain to prove

$$np(min(d^\star), max(d^\star)) = np(min(d), max(d)) + 1, \tag{19.1}$$

$$pp(min(d^\star), max(d^\star)) = pp(min(d), max(d)) \mathbin{+\!\!+} [np(min(d), max(d))]. \tag{19.2}$$

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

$$min(d^\star) = min(d), \tag{20.1}$$

$$max(d^\star) = max(d) + 1. \tag{20.2}$$

For the proof of (20.1), consider the following computation

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

For the proof of (20.2), consider the following computation

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

Now these propositions are proved, we can use them while proving (19.1) and (19.2). For the proof of (19.1), 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 (19.2), consider the following computation

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

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

$$np(min(d^\star), max(d^\star)) = np(min(d), max(d)), \tag{21.1}$$

$$pp(min(d^\star), max(d^\star)) = drop(pp(min(d), max(d)), s). \tag{21.2}$$

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

$$min(d^\star) = min(d) + s, \tag{22.1}$$
$$max(d^\star) = max(d). \tag{22.2}$$

For the proof of (22.1), consider the following computation

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

For the proof of (22.2), 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 (21.1) and (21.2). For the proof of (21.1), 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 (21.2), 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 3. *For any batch size s, we have $\Xi = \Psi$.*

Proof: This follows immediately from Theorem 1 and the fact that h, as defined in Definition 9, is a state mapping (satisfies the matching criteria). □

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 a particular formal method.

Before we draw conclusions concerning these matters, let us first conclude the following on the use of χ, μCRL, 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 μCRL 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 main issue when translating a χ model into its μCRL equivalent is that one has to take into account a second programming paradigm. The χ language uses an imperative approach for both the process part and the arithmetic part, while μCRL uses an imperative and a declarative approach respectively. Furthermore, in χ the process part and arithmetic part are mixed while μCRL has a clear distinction between both parts.
- 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.
- One of the disadvantages of the approach towards verification used in this paper is that the invariants, as listed in Table 9, do not correspond well with intuition.
- 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 μCRL model and because we may, on intuitive grounds, conclude that we succeeded in preserving the semantics while deriving the μCRL model from the χ model, we allow ourselves to apply the statements to be made on the μCRL model to the χ model. Current research is concerned with enabling direct formal reasoning about χ models and enabling formal translations to other specification languages, like μCRL, by setting up a formal semantics of χ [8].

We succeeded in giving a correctness proof by proving that the external behaviour of the implementation is equivalent with the specification ($\Xi = \Psi$). 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 of industrial systems.

Furthermore, it should decrease the time needed to come to correct design of industrial systems.

Although, for the case study presented in this article, the respective steps of the proof methodology have been carried out manually, many of these steps can be performed mechanically.

- The translation of the χ-models into μCRL processes can at the moment not be supported mechanically. As soon as the intuitions underlying the translation presented in this paper are captured in a formal semantics, this topic can be addressed.
- The linearisation of the parallel composition of the μCRL processes describing the χ-models can be performed mechanically [19].
- Given linear process operators for the specification and implementation of a system, the conditions that have to be satisfied according to the cones and foci theorem (i.e. the matching criteria) can easily be generated automatically.
- Checking whether a given formula is an invariant for a given linear process operator can be mechanically supported by theorem provers such as PVS [32] and HOL [16].

Besides these, the μCRL toolset offers a state space generator that interfaces well with the Caesar/Aldebaran toolset [15], a simulator and several tools to symbolically reduce a generated state space [41].

Acknowledgments

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

References

1. W. Alberts, and G. Naumoski, "A discrete-event simulator for systems engineering," Ph.D. thesis, Eindhoven University of Technology, 1998.
2. M. Andersson, "Object-oriented modeling and simulation of hybrid systems," Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, 1994.
3. N. Arends, "A systems engineering specification formalism," Ph.D. thesis, Eindhoven University of Technology, 1996.
4. H. Barendrecht, *The Lambda Calculus, its Syntax and Semantics (revised ed.)*, Vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
5. J. Bergstra and J. Klop, "Process algebra for synchronous communication," *Information and Control*, Vol. 60, pp. 109–137, 1984.
6. M. Bezem, R. Bol, and J. Groote, "Formalizing process algebraic verifications in the calculus of constructions," *Formal Aspects of Computing* Vol. 9, pp. 1–48, 1997.
7. M. Bezem and J. Groote, "A correctness proof of a one-bit sliding window protocol in μ*CRL*," *The Computer Journal*, Vol. 37, No. 4, pp. 289–307, 1994.
8. V. Bos and J. Kleijn, "Structured operational semantics of χ," Computing Science Reports 99/01, Eindhoven University of Technology, Eindhoven, The Netherlands, 1999.
9. K. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1989.
10. A. Dewey, *Analysis and Design of Digital Systems with VHDL*, Brooks/Cole, 1997.

11. E. Dijkstra, *A Discipline of Programming*, Prentice-Hall Series in Automatic Computation, Prentice-Hall, 1976.
12. G. Fábián, "A language and simulator for hybrid systems," Ph.D. thesis, Eindhoven University of Technology, The Netherlands, 1999.
13. J. Fey, "Design of a fruit juice blending and packaging plant," Ph.D. thesis, Eindhoven University of Technology, The Netherlands, to appear.
14. L.-Å. Fredlund, J. Groote, and H. Korver, "Formal verification of a leader election protocol in process algebra," *Theoretical Computer Science*, Vol. 177, pp. 459–486, 1997.
15. H. Garavel, M. Jorgensen, R. Mateescu, C. Pecheur, M. Sighireanu, and B. Vivien, "CADP'97—Status, applications, and perspectives," in I. Lovrek (Ed.), *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design*, Zagreb, Croatia, 1997.
16. M. Gordon and T. Melham, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.
17. J. Groote and A. Ponse, "Proof theory for μCRL: A language for processes with data," in D. Andrews, J. Groote, and C. Middelburg (Eds.), *Proceedings of the International Workshop on Semantics of Specification Languages*. The Netherlands, 1994, pp. 231–250,
18. J. Groote and A. Ponse, "The syntax and semantics of μCRL," in A. Ponse, C. Verhoef, and S. van Vlijmen (Eds.), *ACP: Algebra of Communicating Processes*, Utrecht, The Netherlands, 1995, pp. 26–62.
19. J. Groote, A. Ponse, and Y. Usenko, "Linearization in parallel p*CRL*," Report SEN-R0019, CWI, 2000.
20. 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, 1995.
21. J. Groote and J. Springintveld, "Algebraic verification of a distributed summation algorithm," Technical Report R9640, CWI, Amsterdam, 1996.
22. J. Groote and J. van de Pol, "A bounded retransmission protocol for large data packets. A case study in computer checked verification," in M. Wirsing and M. Nivat (Eds.), *Proceedings of AMAST'96*, Vol. 1101 of *Lecture Notes in Computer Science*, Munich, 1996, pp. 536–550.
23. J. Groote and J. van Wamel, "Analysis of three hybrid systems in timed μCRL," Report SEN-R9815, CWI. To appear in Science of Computer Programming, 1998.
24. P. Haagh, A. Wilkens, H. Rulkens, E. van Campen, and J. Rooda, "Application of a layout design method to the dielectric decomposition area in a 300 mm wafer fab," in *Proceedings of the Seventh International Symposium on Semiconductor Manufacturing*. Tokyo, Japan, 1998, pp. 69–72.
25. C. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985
26. IEEE, "IEEE Standard VHDL Language Reference Manual/Sh14894," IEEE standards interpretations edition, 1987. IEEE Std 1076-1987.
27. D. Kettenis, "Issues of parallelization in implementation of the combined simulation language COSMOS," Ph.D. thesis, Delft University of Technology, 1994.
28. B. Khoshnevis, *Discrete Systems Simulation*, McGraw-Hill, 1994.
29. H. Korver and M. Sellink, "On automating process algebra proofs," in V. Atalay, U. Halici, K. Inan, N. Yalabik, and A. Yazici (Eds.), *Proceedings of the Eleventh International Symposium on Computer and Information Sciences (ISCIS XI)*, Antalya, Turkey, 1996, pp. 815–826.
30. N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.
31. E. Mitchell and J. Gauthier, "Advanced continuous simulation language," *Simulation*, Vol. 26, No. 3, pp. 72–78, 1976.
32. S. Owre, J.M. Rushby, and N. Shankar, "PVS: A prototype verification System," in D. Kapur (Ed.), *11th International Conference on Automated Deduction (CADE)*, Vol. 607 of *Springer Verlag Lecture Notes in Artificial Intelligence*. Saratoga, NY, 1992, pp. 748–752 (reprint).
33. C. Pegden, R. Shannon, and R. Sadowski, *Introduction to Simulation Using SIMAN*, McGraw-Hill, 1995.
34. J. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.
35. C. Roth, *Digital Systems Design Using VHDL*, Brooks/Cole, 1998.
36. H. Rulkens, E. van Campen, J. van Herk, and J. Rooda, "Batch size optimization of a furnace and pre clean area by using dynamic simulations," in *Proceedings of the Advanced Semiconductor Manufacturing Conference*. Boston, 1998, pp. 439–444.

37. E. van Campen, "Design of a multi-product, multi-process wafer fab," Ph.D. thesis, Eindhoven University of Technology, The Netherlands. To appear.
38. J. van de Mortel-Fronczak and J. Rooda, "Heterarchical control systems for production cells—A case study," in *Proceedings of MIM'97*. Vienna, Austria, 1997, pp. 243–248.
39. K. van Hee, L. Somers, and M. Voorhoeve, "The EXSPECT tool," in S. Prehn and W. Toetenel (Eds.), *VDM'91—Formal Software Development*, Vol. 551 of *Lecture Notes in Computer Science*. 1991, pp. 683–684.
40. J. van Wamel, "Verification techniques for elementary data types and retransmission protocols," Ph.D. thesis, University of Amsterdam, 1995.
41. A. Wauters, "Manual for the μCRL toolset: Version 1.11," 2000. Available at http://www.cwi.nl/˜mcrl.