

Suitability of mCRL2 for Concurrent-System Design: A 2×2 Switch Case Study

Frank P.M. Stappers, Michel A. Reniers, and Jan Friso Groote

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

Abstract. Specifying concurrent systems can be done using a variety of languages. These languages have different features and therefore are not necessarily equally suitable for capturing concepts from reality with respect to both expressivity and ease-of-use.

This paper addresses these aspects for the specification language mCRL2 by considering the 2×2 Switch case study. This case study has been used before to compare other specification languages, more specifically TLA+, Bluespec, Statecharts and ACP. The case study primarily focuses on two important features, namely multi-party communication and priority of certain actions over other actions. We show that mCRL2 is appropriate for the specification of these features, especially multi-party communication. Moreover, we express some of the requirements of the original case study in terms of modal μ -calculus formulae and establish that these are indeed satisfied by the model.

1 Introduction

In today's world, there are many different ways to specify system's behavior. At first, many specification languages seem suitable for describing system behaviour, as they are applied to case studies and toy examples that are specially tailored to assess certain features of a language. Unfortunately, when actual systems need to be specified, it often turns out that a language cannot express a certain amount of behavior, as the language is too generic or too limited, and therefore not vigorous enough to express complex behavioral patterns. This way, designers are required to deviate from the system's behaviour or they have to apply abstractions such that inexpressible behavior becomes irrelevant.

When designs are finished, it is difficult to ensure that a system meets the requirements that were agreed upon in advance. In many cases human reasoning is applied to validate that a system meets these requirements. However, a proof or guarantee cannot be given. Especially for mission critical systems, but also for concurrent systems, this might yield to undesired behaviour, which can result into catastrophic disasters.

Selecting a suitable language for system design is a difficult task. To guide designers, the authors of [6] have recently compared the specification languages TLA+, Bluespec, Statecharts, and ACP for a particular case study. The authors of [6] compare these languages with respect to the following three criteria:

1. the *local* (as opposed to global and temporal) reasoning that is required by the designer in order to specify behaviour,
2. *adaptability* to variations in design intent, and
3. checking whether a specification *captures* the corresponding design intent.

The case study they selected, deals with a switch that internally routes packets from input buffers to output buffers. These packets are routed according to a set of rules that specify priority amongst selected packets as well as simultaneous packet transfers. As these rules are complementary to each other, they illustrate contradictive concerns and emphasize on the possible weaknesses of the specification languages. In [6] it is concluded that each of the used specification languages performs poorly for at least two of these criteria.

In extension to the framework, presented in [6], this paper puts mCRL2 [9] to the same test. The goal of this paper is to show that the specification language mCRL2 is better suited than the other specification languages, at least for the presented case study.

mCRL2 is a specification language, especially targeted for describing communication behaviour among systems. The behavioural part of the language is based on process algebra [1]. For the purpose of specifying behaviour, mCRL2 facilitates a data part which is based on higher-order abstract equational data types. It allows quantifiers, (unbounded) integers, (infinite) sets and bags, structured types, lists and real numbers, that are set up as close as possible to their mathematical counterparts.

The models that we present for the cases are obtained in a relatively straightforward way from the informal description. It turns out that multi-party communication is easily captured by the advanced communication mechanisms of mCRL2. mCRL2 has no direct support for specifying priority. Nevertheless we are able to describe the types of priority used in the cases at hand.

For the manipulation, analysis and visualisation of specifications, the language is equipped by a range of tools [7,10]. These tools allow amongst others the verification of requirements that are described in the modal μ -calculus [13].

This paper is structured as follows. Section 2 gives a brief introduction to the relevant fragments of the language mCRL2 and the modal μ -calculus. The switches are modelled in Sections 3,4 and 5. Section 6 elaborates on the requirements that have been verified on the constructed models. Section 7 compares the work presented here to that of others. Section 8 describes our conclusions and future work.

2 Preliminaries

2.1 Syntax and Semantics of mCRL2

An mCRL2 process is built from data-parameterized multi-actions and a collection of process operators. In this paper, a fragment of the syntax of the un-timed mCRL2 language is used. It is given by the following *BNF*:

$$\begin{aligned}
P &::= \alpha \mid P + P \mid P \cdot P \mid c \rightarrow P \mid \sum_{x:D} P \mid P \parallel P \\
&\mid \partial_B(P) \mid \tau_B(P) \mid \Gamma_V(P) \mid X(\mathbf{d}) \\
\alpha &::= \tau \mid a(\mathbf{d}) \mid \alpha \mid \alpha
\end{aligned}$$

The small \mid indicates a choice between symbols in the expression of the BNF. In this syntax α denotes a multi-action. A multi-action consists of actions combined by the big \mid . The empty multi-action is denoted by τ . An action $a(\mathbf{d})$ consists of an action name a and possibility a data parameter vector \mathbf{d} (the syntax of which is left unspecified). A multi-action represents the simultaneous execution of the constituent actions.

Processes are denoted by P . For processes, $+$ denotes non-deterministic choice, i.e., a choice between behaviors, \cdot denotes sequential composition, i.e., a process followed by another process. The conditional operator, written as $c \rightarrow p$, denotes that if c data expression of sort \mathbb{B} holds, then process P is executed. The non-deterministic choice among processes is denoted by $\sum_{x:D} P$, where x is a variable of sort D and P is a process expression in which the variable x may occur. The parallel composition of processes is represented by \parallel operator, that denotes the concurrent execution of both processes. The operator ∂_B blocks all actions from set B of action names, i.e., prevents the occurrence of the specified actions. The operator τ_B replaces all occurrences of actions from B by τ . Γ_V applies the communications described by the set V to a process. A communication in the set V is of the form $a_1 \mid \dots \mid a_n \rightarrow a$. Application of Γ_V to a process means that any occurrence of the multi-action $a_1(\mathbf{d}) \mid \dots \mid a_n(\mathbf{d})$ is replaced by $a(\mathbf{d})$, for any \mathbf{d} . $X(\mathbf{d})$ is a reference to a process definition of the form $X(\mathbf{x}) = P$, i.e., the process $X(\mathbf{d})$ behaves as prescribed by P with \mathbf{x} replaced by \mathbf{d} .

The semantics associated with an mCRL2 process, as used in the mCRL2 tool set, is a transition system where the transitions are labelled by multi-actions. A more elaborate description of the syntax and (timed) semantics are given in [9,10].

2.2 Modal μ -Calculus

Modal μ -calculus formulae are used to describe behavioral properties. These properties are verified against a behavioral model described in mCRL2. In this paper, requirements are specified in a variant of the modal μ -calculus extended with regular expressions [8] and data. The restricted fragment of the modal μ -calculus used, is as follows:

$$\begin{aligned}
\phi &::= \text{false} \mid \phi \Rightarrow \phi \mid \phi \wedge \phi \mid [\rho]\phi \mid \langle \rho \rangle \phi \mid \forall_{x:D} \phi \mid c \\
\rho &::= \alpha \mid \rho \cdot \rho \mid \rho^* \\
\alpha &::= a(\mathbf{d}) \mid \neg \alpha \mid \alpha \mid \alpha \mid \text{true}
\end{aligned}$$

In this syntax, ϕ represents a property, ρ represents a set of sequences of actions and α represents the absence or presence of a multi-action. An arbitrary multi-action is denoted by *true*. The property *false* holds for no model. The property $[\rho]\phi$ states the property that ϕ holds in all states that can be reached by a

sequence described by ρ . The property $\langle \rho \rangle \phi$ describes that ϕ holds in some state that can be reached by a sequence from ρ . To describe action sequences concatenation and iteration can be used. A more elaborate description of the modal μ -calculus and its semantics can be found in [5,8].

3 Specification of the Simple 2×2 Switch

The 2×2 Switch case study consists of three separate cases that gradually increase in difficulty. These cases are referred to as the “Simplified Switch”, the “Original Switch” [4] and the “Modified Switch”. In the specification of the three cases, we follow the informal description from [6] as closely as possible. This means that we introduce a single process for each of the four buffers. By means of the advanced communication mechanisms offered in mCRL2, we describe their non-trivial interaction. In this section, and in the sections to follow, we discuss the way in which we have dealt with the modeling challenges posed by the case studies.

The Simplified Switch contains two input FIFO buffers and two output FIFO buffers. All buffers have a unique identity, w.r.t. the type of buffer, e.g. each input or output buffer corresponds to a numerical value, and a finite capacity for storing packets. All buffers have the same capacity.

Each packet consists of 32 bits. Packets enter the system via the input buffers and depart the system via the output buffers. Packets are transferred from an input buffer to one of the output buffers based on the first bit of the packet: If the first bit of a packet is 0, it is routed to the output buffer with identity 0, and otherwise it is routed to the output buffer with identity 1.

The packets may only be transferred if the relevant output buffer is not full. A buffer operates per clock cycle and can do at most one operation, namely receive a packet, send a packet, or nothing. Furthermore, we require maximum throughput, e.g. a packet should be transferred if it has the ability to. Next to that, if packets from different input buffers are available for transferral to the same output buffer, transferral of the packet from input buffer 0 gets priority over transferral of the packet from input buffer 1.

3.1 Bits and Packets

The data type of bits consists of two different values. In mCRL2, this is defined as:

```
sort Bit = struct zero | one;
```

In the case study, packets consist of 32 bits. This implies that a single packet can be represented by 2^{32} different configurations. mCRL2 allows the description of such a data type without any problems; e.g., by a structured sort that composes 32 bits by:

```
sort Packet = struct packet( $b_1, b_2, \dots, b_{32} : \textit{Bit}$ );
```

From a modelling point of view, we do not object to such a representation or see any difficulty to write it down in an mCRL2 specification. Unfortunately, for a formal analysis with tools that require an explicit state space generation such as model-checking tools, this has an apparent drawback. It gives rise to 2^{32} different potential contents for each position in each of the considered buffers. This number is usually too big to be handled by current state-of-the-art model-checking tools. For that reason we require an appropriate abstraction.

Investigation shows that only two types of data packets are relevant for the Simplified Switch. First, those data packets for which the first bit of the packet is 0, and second, those data packets for which the first bit is 1. According to the first bit, packets are respectively routed to output buffer 0 or to output buffer 1. For this reason, we choose to abstract from the irrelevant bits of a packet, by only modeling the first bit. Consequently, the structure of a packet is redefined as:

```
sort Packet = struct packet( $b_1$  : Bit);
```

To route packets, we require a function that assigns a destination to a given packet. So, we define a mapping *dest* that expresses the relation between the data within the packet and the output buffer to which the packet is to be routed.

```
map dest : Packet  $\rightarrow$   $\mathbb{N}$ ;
eqn dest(packet(zero)) = 0;
      dest(packet(one)) = 1;
```

3.2 Capacity of the Buffers

The system consists of four queues. Each buffer has the same capacity *cap*, which is assumed to be at least 1. In order to specify the case study without referring to an explicitly defined value we introduce the following constant.

```
map cap : Pos;
```

By means of an equation we may assign a specific value to this mapping. This is necessary for state space generation and simulation of the specification. This way changing the capacity, if desired, needs to be done in one place only.

```
eqn cap = 3;
```

3.3 Information Exchange between the Processes

To observe packets that enter and leave the 2×2 -switch, two parameterized actions are introduced, namely one for adding an element to an input buffer (*enter*) and another one for removing an element from an output buffer (*leave*). The first data parameter refers to the identity of an input buffer (for *enter*-actions) or an output buffer (in case of *leave*-actions). The second data parameter is used to represent the actual data for the packet itself.

act *enter* : $\mathbb{N} \times \text{Packet}$;
leave : $\mathbb{N} \times \text{Packet}$;

The sending of a packet from an input buffer to an output buffer is described by means of the *send* action. Similarly, for the receipt of a packet by an output buffer, the action *recv* is used. To synchronize actions, mCRL2 provides synchronous communication between processes, if all the action data parameters in the synchronizing actions have the same value. To show (and observe) that a send and receive synchronize, we use the action *comm*, which reflects the successful synchronization of a *send* and a *recv*.

The actions *send*, *recv* and *comm* are each modeled with three data parameters. The first parameter is used to denote the identity of the input buffer that sends the package, the second parameter denotes the identity of the output buffer that receives the package, and the last parameter denotes the packet that is actually being transferred. The first and second parameter provide handles to observe the routing of packets; i.e., they are used to express and verify requirements later on. The last data parameter is required to transfer and observe the data flow between buffers. Note that the second parameter is a cosmetic addition, as its can also be obtained from the data of the packet itself.

act *send* : $\mathbb{N} \times \mathbb{N} \times \text{Packet}$;
recv : $\mathbb{N} \times \mathbb{N} \times \text{Packet}$;
comm : $\mathbb{N} \times \mathbb{N} \times \text{Packet}$;

In the Simplified Switch case study, the packet exchange between an input buffer, say *i*, and an output buffer, say *o*, not only depends on the behavior expressed in the processes, but also on the contents of the other input buffer. In mCRL2, it is possible to use multi-party communication to establish the involvement of another process. This means that we require actions that reveal information about a third party in the communication. We introduce actions *grant* and *free* for this purpose. Both *grant*(*i*, *j*, *p*) and *free*(*i*, *j*, *p*) denote that input buffer *i* is granted permission to send a packet *p* to output buffer *j*. One of these actions is used for establishing priority and the other one for simultaneous packet transfer. A more detailed explanation is provided later in this section.

act *grant* : $\mathbb{N} \times \mathbb{N} \times \text{Packet}$;
free : $\mathbb{N} \times \mathbb{N} \times \text{Packet}$;

3.4 The Output Buffers with Capacity *cap*

In mCRL2, a FIFO buffer *Output* with capacity *cap* is given by the following process specification:

proc *Output*(*i* : \mathbb{N} , *c* : *List*(*Packet*)) =
 $\#c < \text{cap} \rightarrow \sum_{s:\mathbb{N}} \sum_{p:\text{Packet}} \text{recv}(s, i, p) \cdot \text{Output}(i, p \triangleright c)$
 $+ c \neq [] \rightarrow \text{leave}(i, \text{rhead}(c)) \cdot \text{Output}(i, \text{rtail}(c));$

The first line in the above model specifies the name of the process and declares the associated process parameters. In this case the buffer has two parameters. The first process parameter represents the identity of an output buffer. The second process parameter captures the contents of the queue as a list of packets. As already described, an arbitrary packet can be received as long as the buffer is not yet full ($\#c$ denotes the number of elements in the list c). So the first summand, specifies that when a packet is received, it is appended to the buffer. Appending a packet p to a buffer contents c is denoted by $p \triangleright c$. The second summand describes that the packet (if any) that has been inserted into the queue (so the buffer is not empty, $c \not\approx []$) first ($rhead(c)$ denotes this element), it exits the switch by means of the *leave* action and is removed from the queue ($rtail(c)$). Note that modeling the buffer in this way, the specification of the output buffer does not rely in any way on the fact that only packets with a specific first bit will be sent to it, e.g. it accepts packets regardless of their content.

For both the output buffer as described in this subsection and the input buffer described in the following subsection we have chosen to allow it to perform at most one action at the same time.

3.5 The Input Buffers with Capacity cap

The main challenges of this modeling exercise are to deal appropriately with the priority of input buffer 0 over input buffer 1 in case both buffers want to transfer a packet to the same destination; and to deal with the required simultaneous packet transfer in case both buffers want to transfer packets to different destinations. In this section we gradually shape the model, by defining the interaction between the different processes, as well as specifying the input buffer such that it eventually complies to the settled design intent.

We start by modeling the behavior of the input buffer analogously to the output buffer:

```

proc Input( $i : \mathbb{N}, c : List(Packet)$ ) =
     $\#c < cap \rightarrow \sum_{p:Packet} enter(i, p) \cdot Input(i, p \triangleright c)$ 
    +  $c \not\approx [] \rightarrow send(i, dest(rhead(c)), rhead(c)) \cdot Input(i, rtail(c));$ 

```

Next, we setup the basic communication between input buffers and output buffers. We first specify that the four buffers require to run in parallel. Furthermore, we specify that a successful synchronization of *send* and *recv* actions, results in a *comm* action. This is expressed by means of the subscript parameter $send|recv \rightarrow comm$ in the communication operator Γ . We only allow successful communications, therefore we encapsulate all *send* and *recv* actions that do not result in a successful synchronization. This way, insertion or removal of a packet can be done simultaneously, while other buffers transfer packets. Combining the instantiated process definitions with the communication and encapsulated operators, leads to the following initialization:

init $\partial_{\{send,recv\}}(\Gamma_{\{send|recv \rightarrow comm\}}(\text{Input}(0, \square) \parallel \text{Input}(1, \square) \parallel \text{Output}(0, \square) \parallel \text{Output}(1, \square)))$;

To acquire the simultaneous packet transfer and prioritized packet transfer, the model needs to be adapted in two ways. The first step takes care of the prioritized packet transfer if packets route to the same destination. The second step takes care of the required simultaneous packet transfer to different output buffers.

Prioritized packet transfer. The way in which we deal with the prioritized packet transfer is as follows. The input buffer signals which transfers are allowed for execution by the other input buffer by means of the *grant*-action. If a buffer is empty it grants permission for any transfer in the other process of the input queue. If the buffer is not empty it only grants permission for a transferral of packets from each input buffer with a lower identity to the same output buffer.

proc $\text{Input}(i : \mathbb{N}, c : \text{List}(\text{Packet})) =$
 $\#c < cap \rightarrow \sum_{p:\text{Packet}} \text{enter}(i, p) \cdot \text{Input}(i, p \triangleright c)$
 $+ c \not\approx \square \rightarrow \text{send}(i, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) \cdot \text{Input}(i, \text{rtail}(c));$
 $+ c \approx \square \rightarrow \sum_{n,m:\mathbb{N}} \sum_{p:\text{Packet}} \text{grant}(n, m, p) \cdot \text{Input}(i, c)$
 $+ c \not\approx \square \rightarrow \sum_{n:\mathbb{N}} n < i \rightarrow \text{grant}(n, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) \cdot \text{Input}(i, c)$

To ensure that the *grant*-action synchronizes with the other corresponding *send*- and *recv*-actions another communication function is added:

init $\partial_{\{send,recv,grant\}}(\Gamma_{\{send|recv \rightarrow comm\}}(\Gamma_{\{send|recv|grant \rightarrow comm\}}(\text{Input}(0, \square) \parallel \text{Input}(1, \square) \parallel \text{Output}(0, \square) \parallel \text{Output}(1, \square))))$;

The nesting of the communication functions this way is necessary to ensure that priority is given.

Maximal communication. In order to meet the second requirement, the input buffer announces that it allows a simultaneous transferral of packets (from the other input buffer) with a different destination via the *free*-action.

proc $\text{Input}(i : \mathbb{N}, c : \text{List}(\text{Packet})) =$
 $\#c < cap \rightarrow \sum_{p:\text{Packet}} \text{enter}(i, p) \cdot \text{Input}(i, p \triangleright c)$
 $+ c \not\approx \square \rightarrow \sum_{n:\mathbb{N}} \sum_{p:\text{Packet}} n \not\approx i \wedge \text{dest}(p) \not\approx \text{dest}(\text{rhead}(c)) \rightarrow$
 $\text{send}(i, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) | \text{free}(n, \text{dest}(p), p) \cdot \text{Input}(i, \text{rtail}(c))$
 $+ c \not\approx \square \rightarrow \text{send}(i, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) \cdot \text{Input}(i, \text{rtail}(c))$
 $+ c \approx \square \rightarrow \sum_{n,m:\mathbb{N}} \sum_{p:\text{Packet}} \text{grant}(n, m, p) \cdot \text{Input}(i, c)$
 $+ c \not\approx \square \rightarrow \sum_{n:\mathbb{N}} n < i \rightarrow \text{grant}(n, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) \cdot \text{Input}(i, c);$

By adapting the communications in the outermost communication operator to $\{send|recv|free \rightarrow comm\}$ we achieve that packet transfers are only allowed in case the other input buffer grants permission. This way simultaneous packet transfers are achieved whenever possible. All possible communications are now permitted by either a *grant*- or a *free*-action.

init $\partial_{\{send,recv,grant,free\}}(\Gamma_{\{send|recv|grant \rightarrow comm\}}(\Gamma_{\{send|recv|free \rightarrow comm\}}(Input(0, []) \parallel Input(1, []) \parallel Output(0, []) \parallel Output(1, [])))));$

The order in which the communication operators are applied to the parallel composition of the buffers is of no importance. It is not allowed to declare both communication operators by means of one communication operator since the left-hand sides of the communication patterns share an action, which might lead to a non-unique solution. For that reason the communication operators are placed in a hierarchical composition. We conjecture that the order of these communication operators is of no importance. To provide (partial) evidence, we have validated this claim, by using the mCRL2 tool set to establish that the respective labelled transition systems are strongly bisimilar (even isomorphic) for the case that the capacity of the buffers is 1, 2 and 3.

4 Specification of the Original 2×2 Switch

The Original Switch is an extension of the Simplified Switch. The Original switch contains an additional counter, that counts interesting packets that are transferred from input to output buffers. A packet is considered interesting if its second, third, and fourth bit are all 0. The counter is restricted. Therefore the value can only be incremented by one per clock cycle. So when both input buffers are capable of transferring interesting packets, priority is given to the transferral of packets from input buffer 0 and the transferral of packets from input buffer 1 is delayed. Thus, we may only transfer packets simultaneously if they are not both interesting. In all other cases a process needs to either take or grant priority as in the Simplified Switch case study.

In this section, we adapt the model of the Simplified Switch to obtain a model that corresponds to the design intent of the Original Switch. Thereto, we need to extend a part of the data specification and adapt the behaviors of the buffer processes slightly.

4.1 Packets

The fact that the second, third and fourth bit of a packet have become relevant for the behaviour of the switch means that we have to reconsider our definition of the data type representing packets. We can introduce packets with four bits (all relevant ones) in a way similar to the current definition. Instead, and more abstractly, we decide to model packets as before but now with an additional

Boolean parameter indicating whether the packet is interesting (*true*) or not (*false*).

```
sort Packet = struct packet(b1 : Bit, int :  $\mathbb{B}$ );
```

By extending the structured sort, we are required to update the definition of the mapping for routing packets. As the second, third, and fourth bit have no effect on the destination of a packet, the adaptation is straightforward.

```
map dest : Packet →  $\mathbb{N}$ ;
var b :  $\mathbb{B}$ ;
eqn dest(packet(zero, b)) = 0;
     dest(packet(one, b)) = 1;
```

4.2 The Act of Counting

There are several ways of modelling the counting of interesting packets. One way is to introduce a parameter that reflects the number of interesting packets that have been transferred. Another way is to introduce an action to indicate a transferral of an interesting packet. We have chosen the latter solution. Thus, the act of counting interesting packets will be performed by executing an action *inc*, that has no data parameters.

```
act inc;
```

Another decision that must be made is which entity actually performs the counting. One solution is to introduce a separate process for this purpose. Another option is to enhance the functionality of either the input or the output buffers. We have chosen to enhance the functionality of the output buffers. It should be said that implementing the other solutions poses no real problems for mCRL2.

To accommodate this behavior, the first summand of the output buffer from the Simplified Switch is split into two cases, one for receiving and counting an interesting packet and one for receiving a non-interesting packet. To decide if a packet is interesting, the projection function *int* is used. The projection function for a specific field of a structured sort is specified in the sort declaration. For the sort *Packet* there are projection functions *b*₁ and *int* for obtaining the values of the first and second field, respectively.

```
proc Output(i :  $\mathbb{N}$ , c : List(Packet)) =
  #c < cap →  $\sum_{s:\mathbb{N}}$   $\sum_{p:\mathit{Packet}}$  (int(p) → recv(s, i, p)inc · Output(i, p ▷ c)
  +  $\neg$ int(p) → recv(s, i, p) · Output(i, p ▷ c)
  + c  $\not\approx$  [] → leave(i, rhead(c)) · Output(i, rtail(c));
```

4.3 Adapting the Input Buffer

The Original Switch poses an additional restriction on the cases in which communication between input and output buffer can be performed.

We may only transfer packets simultaneously if they have different destinations and at most one packet is interesting. This is expressed in the second summand below.

In case both input buffers contain an interesting packet and these packets have different destinations, priority is granted to any input buffer with lower identity. See the fifth summand below.

We are required to grant priority to both interesting and non-interesting packets if the local packet is non-interesting. For that reason, the last summand is adapted as well.

$$\begin{aligned}
\mathbf{proc} \text{ Input}(i : \mathbb{N}, c : \text{List}(\text{Packet})) = & \\
\#c < \text{cap} \rightarrow \sum_{p:\text{Packet}} \text{enter}(i, p) \cdot \text{Input}(i, p \triangleright c) & \\
+ c \not\approx \square \rightarrow & \\
\sum_{p:\text{Packet}} \text{dest}(p) \not\approx \text{dest}(\text{rhead}(c)) \wedge (\neg \text{int}(p) \vee \neg \text{int}(\text{rhead}(c))) \rightarrow & \\
\sum_{n:\mathbb{N}} n \not\approx i \rightarrow \text{send}(i, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) | \text{free}(n, \text{dest}(p), p) \cdot & \\
\text{Input}(i, \text{rtail}(c)) & \\
+ c \not\approx \square \rightarrow \text{send}(i, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) \cdot \text{Input}(i, \text{rtail}(c)) & \\
+ c \approx \square \rightarrow \sum_{n,m:\mathbb{N}} \sum_{p:\text{Packet}} \text{grant}(n, m, p) \cdot \text{Input}(i, c) & \\
+ c \not\approx \square \wedge \text{int}(\text{rhead}(c)) \rightarrow \sum_{p:\text{Packet}} \text{dest}(p) \approx \text{dest}(\text{rhead}(c)) \vee \text{int}(p) \rightarrow & \\
\sum_{n:\mathbb{N}} n < i \rightarrow \text{grant}(n, \text{dest}(p), p) \cdot \text{Input}(i, c) & \\
+ c \not\approx \square \wedge \neg \text{int}(\text{rhead}(c)) \rightarrow \sum_{p:\text{Packet}} b_1(p) \approx b_1(\text{rhead}(c)) \rightarrow & \\
\sum_{n:\mathbb{N}} n < i \rightarrow \text{grant}(n, \text{dest}(\text{rhead}(c)), p) \cdot \text{Input}(i, c); &
\end{aligned}$$

5 Specification of the Modified 2×2 Switch

The modified 2×2 switch alters the way the priority is handled amongst colliding transfers in the case that the packets are both interesting and have a different destination. We have defined two conditions, namely both head packets have the same destination (C1) and both head packets are interesting (C2). If either one of these conditions holds, priority is given to the transferral of the packet from input buffer 0.

Now, in the Modified Switch, we keep that if C1 holds, the first input buffer will be given priority over the second buffer. However if C1 does not hold, while C2 holds, priority is given to transferral of the packet from input buffer 1.

This only requires the adaptation of the model of the input buffers. In the relevant case this time priority is granted to the input buffer with the higher identity. The last but one summand of the specification of the input buffer of the Original Switch is split in these two cases.

$$\begin{aligned}
\mathbf{proc} \text{ Input}(i : \mathbb{N}, c : \text{List}(\text{Packet})) = & \\
\#c < \text{cap} \rightarrow \sum_{p:\text{Packet}} \text{enter}(i, p) \cdot \text{Input}(i, p \triangleright c) & \\
+ c \not\approx \square \rightarrow & \\
\sum_{p:\text{Packet}} \text{dest}(p) \not\approx \text{dest}(\text{rhead}(c)) \wedge (\neg \text{int}(p) \vee \neg \text{int}(\text{rhead}(c))) \rightarrow & \\
\sum_{n:\mathbb{N}} n \not\approx i \rightarrow \text{send}(i, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) | \text{free}(n, \text{dest}(p), p) \cdot & \\
\text{Input}(i, \text{rtail}(c)) & \\
+ c \not\approx \square \rightarrow \text{send}(i, \text{dest}(\text{rhead}(c)), \text{rhead}(c)) \cdot \text{Input}(i, \text{rtail}(c)) & \\
+ c \approx \square \rightarrow \sum_{n,m:\mathbb{N}} \sum_{p:\text{Packet}} \text{grant}(n, m, p) \cdot \text{Input}(i, c) & \\
+ c \not\approx \square \wedge \text{int}(\text{rhead}(c)) \rightarrow \sum_{p:\text{Packet}} \text{dest}(p) \approx \text{dest}(\text{rhead}(c)) \rightarrow & \\
\sum_{n:\mathbb{N}} n < i \rightarrow \text{grant}(n, \text{dest}(p), p) \cdot \text{Input}(i, c) & \\
+ c \not\approx \square \wedge \text{int}(\text{rhead}(c)) \rightarrow \sum_{p:\text{Packet}} \text{dest}(p) \not\approx \text{dest}(\text{rhead}(c)) \wedge \text{int}(p) \rightarrow & \\
\sum_{n:\mathbb{N}} n > i \rightarrow \text{grant}(n, \text{dest}(p), p) \cdot \text{Input}(i, c) & \\
+ c \not\approx \square \wedge \neg \text{int}(\text{rhead}(c)) \rightarrow \sum_{p:\text{Packet}} b_1(p) \approx b_1(\text{rhead}(c)) \rightarrow & \\
\sum_{n:\mathbb{N}} n < i \rightarrow \text{grant}(n, \text{dest}(\text{rhead}(c)), p) \cdot \text{Input}(i, c); &
\end{aligned}$$

6 Properties of the Models

In [6], the authors presented their models without any form of formal verification. For Statecharts this already led to a model that did not meet the design intent, according to [6]. Their model contained a flaw when both buffers contain interesting head packets and one of the buffers was full while the other was not. In that case, one packet should be delayed while the other head packet was routed. This however was not covered.

To convince readers that our models capture the design intent, we formulate some requirements and verify that the models satisfy them. These requirements relate to deadlock analysis, overflowing buffers, packet collection and maximum progress. The requirements are expressed in terms of modal μ -calculus formulae. The mCRL2 tool set allows for checking the validity of such formulae on labelled transition systems obtained from mCRL2 models.

6.1 Deadlock Detection

Deadlock is a specific condition that brings the system into a halt, from which it cannot execute any behavior for any future. Deadlock can be caused by various reasons, amongst others due to circular resource dependencies or when processes cannot fulfill their precondition in order to execute an extension.

We claim that all of the presented models are free from deadlock. Deadlock freedom is expressed by the following modal μ -calculus formula:

$$[\text{true}^*]\langle \text{true} \rangle \text{true} \quad (1)$$

6.2 Absence of Overflowing Buffers

We have used the standard mCRL2 type construction of lists for modeling the contents of the buffer. Though the lengths of such lists are not fixed or bound from above, the use in combination with the constant *cap* guarantees that there can be no overflows of the buffers. By means of adding the alternative summand:

$$\#c > cap \rightarrow overflow \cdot Input(i, c)$$

to the input buffers, and the summand

$$\#c > cap \rightarrow overflow \cdot Output(i, c)$$

to the output buffers, we can easily check that this situation can never occur by verifying the validity of the modal formula

$$[true^* \cdot overflow]false \quad (2)$$

on the model that is obtained after abstracting from all actions besides *overflow*. This formula then expresses that there can be no execution that performs the action *overflow*.

6.3 Absence of Colliding Packets

The property that no simultaneous packet transfers are possible to the same output buffer is specified by means of the following modal μ -calculus formula:

$$\forall_{p,q:Packet} \forall_{i,j,k:l:\mathbb{N}} [true^* \cdot (comm(i, j, p) | comm(k, j, q))]false \quad (3)$$

This formula must be checked on the model after abstraction from all other actions. This means that for the Simplified Switch the following model has been used

$$\mathbf{init} \quad \tau_{\{enter, leave\}} (\partial_{\{send, recv, grant, free\}} (\Gamma_{\{send|recv|grant \rightarrow comm\}} (\Gamma_{\{send|recv|free \rightarrow comm\}} (Input(0, []) \parallel Input(1, []) \parallel Output(0, []) \parallel Output(1, []))));$$

In a similar way, abstract models for Original and Modified Switch can be defined.

It is not allowed to send two interesting packets simultaneously. This is verified by checking the modal μ -calculus formula

$$\forall_{p,q:Packet} \forall_{i,j,k,l:l:\mathbb{N}} (int(p) \wedge int(q)) \Rightarrow [true^* \cdot (comm(i, j, p) | comm(k, l, q))]false \quad (4)$$

on the system where all environmental actions are abstracted from.

Requirement 3 is relevant for all three models discussed in this paper and Requirement 4 is only relevant for the latter two models.

6.4 Maximal Progress

A property we would like to verify is *maximal progress*. In the context of this case study, the property can be phrased as: “It is impossible to transfer a single packet from an input buffer to an output buffer in case a simultaneous packet transfer is possible.” A modal μ -calculus formula that captures this (provided that it is checked on the model after abstraction of environmental actions) is the following:

$$\forall_{p,q:Packet} ([true^*](\langle comm(0, dest(p), p) | comm(1, dest(q), q) \rangle true \Rightarrow ([comm(1, dest(p), p)]false \wedge [comm(1, dest(q), q)]false))) \quad (5)$$

Note that this way of expressing maximal progress does *not* enforce that packet transfer takes priority over environmental actions.

6.5 Verification Results

The requirements have been checked for the all the (relevant) models, for which the buffers have capacity 3. This buffer capacity has been chosen because it still allows for a reasonably fast analysis. The analysis has been conducted with the mCRL2 tool set (Release 2010, January), on an x86-64 GNU/Linux, running kernel 2.6.31.12, with an Intel® Core™ 2 Duo Mobile Processor T9600 and 4GB of RAM.

The results of the formal analysis are captured in Table 1. Requirements that hold w.r.t. a particular model are marked with “✓”. The time that the verification took is also indicated. Requirements that are irrelevant for a specific model are marked with a “-”. It shows that for each of the models all relevant formulae hold. It should be noted that we have not attempted to reduce these numbers as much as possible by using state space reduction techniques.

Table 1. Results of verifying properties on the models

Requirement	Simplified	Original	Modified
1	✓, 3.550s	✓, 5m3.863s	✓, 5m16.921s
2	✓, 3.729s	✓, 7m35.686s	✓, 7m35.202s
3	✓, 3.778s	✓, 4m44.647s	✓, 4m49.101s
4	-	✓, 5m29.906s	✓, 5m39.844s
5	✓, 3.301s	✓, 4m22.232s	✓, 4m33.786s

7 Comparison

This case study originates from work, gathered in [6]. There, the authors discuss the same case studies, described in the specification languages: TLA+, Bluespec, State-charts and ACP. As we have elaborated on the construction of the different models with their underlying design decisions in mCRL2, this section

describes the deviation of the formalisms with respect to the case study. The comparison focusses on three aspects, namely locality of reasoning, adaptability of the language and maximal throughput. Furthermore we extend the focus by taking verification into account.

Before explaining the comparison, we give a brief description of the four languages. First, TLA+ (the Temporal Logic of Actions) is a complete specification language, that uses logic for the specification and reasoning about concurrent and reactive systems. It is designed for writing specifications consisting of non-temporal mathematics with temporal logic and tries to capture a complete system in a single formula [14]. Second, Bluespec [12] is a guarded command language, based on an operation centric description, where the behavior of a system is described as a collection of atomic operations in the form of rules. These rules are defined by a predicate condition and the effect on the state of the system. During execution several rules are concurrently executed in a clock cycle. Third we consider Statecharts, which are an extension of conventional state-transition diagrams with three elements, dealing, with hierarchy, concurrency and communication [11]. The graphical hierarchy presentation enables designers to adapt to the required level of detail of the system. Finally, the comparison covers the Algebra of Communicating Processes (ACP) [3]. ACP is a finite axiomatisation based framework for specifying and manipulating the behaviour of models. It facilitates the behavioural description for non-deterministic choices, sequential operations, parallel composition, deadlock and communication.

7.1 Maximal Throughput

Within the specification maximal throughput is achieved by executing multiple actions in a single clock cycle. Therefore, this comparison narrows down the scope to the behaviour for simultaneous actions.

It is not possible to describe the simultaneous transfer of packets in TLA+ and ACP. Therefore a designer is required to apply a spatial reasoning to verify that indeed packets are transferred simultaneously. As we compare the formalism to mCRL2, we see that within mCRL2, it is possible to define multi-actions. We believe that these multi-actions are more suitable for specifying the throughput behavior as they relate better to the simultaneous packet transfers in the system. Therefore it is not necessary for a designer to reason about multiple transitions.

For Bluespec specifications, a greedy run-time scheduler tries to acquire maximal throughput. It should be noted that in some cases a maximal throughput cannot be obtained, even though all conflict-free rules are selected. To minimize latency, the scheduler may chose a maximal set of actions of the design for execution during each hardware clock cycle. Therefore it is possible that this set does violate the maximal throughput requirement [15]. As exploration in mCRL2 is exhaustive, and latency is no issue, maximal throughput can be guaranteed, by means of synchronizing actions and guards. Furthermore, although not specified here, we believe that it possible to use the mCRL2 time operator to enforce throughput in different ways, e.g. by enforcing the execution of actions at predefined timestamps.

Regarding Statecharts, the authors of [6] did not give a suitable description in their paper, as they specified a wrong model. Therefore a comparison for maximal throughput, renders useless as a throughput analysis on Statecharts is omitted. Note that this does not mean that it is impossible to give a correct model using Statecharts.

7.2 Priority

The locality of reasoning is derived from the way priority is assigned to the routing of packets.

To reduce the amount of global reason, w.r.t. the communication we have generalized from the specific implementations of the input buffers. This allows us to reason on a local level about priorities. This shows if we compare our models to those as given in ACP. Note that within mCRL2, we have modelled priority by means of permissions, and therefore the contents of the buffers are invisible to other processes. In the given ACP models, the queues are directly inspected by the other processes. This requires a more spatial reasoning in ACP, in order to derive the priority.

Within TLA+, the priority of a packet transfer is handled at a local level. So with respect to assigning priority to executing actions, mCRL2 and TLA+ are comparable. We do have to note that the input queues, as well as the output queues are grouped in TLA+. This makes it possible for TLA+ actions to directly observe the queue of another process, at a local level. When comparing this method to the one given in our models, we believe that it is possible in mCRL2 to apply reasoning on a more local level.

The Bluespec specification defines rules that implicitly deal with mutually exclusive access to shared resources. When multiple rules access the same resource, access is given to the resource defined first in the priority hierarchy. In this way, priority amongst packet transfers is ensured. Note that priority rules are defined on a spatial level. Therefore, the reasoning that needs to be applied is more spatial than the one used in mCRL2.

Within Statecharts all the behavior of the buffers are locally specified, however global temporal reasoning is required to establish the priority among packet transfers. A simultaneous transfer requires a global spatial reasoning over at least four individual Statecharts.

7.3 Adaptability

In [6], the authors only explain TLA+ for the simple switch. Though they claim that TLA+ relates to Bluespec, they do not show models for the original and modified switch. For that reason, the adaptability of TLA+ is unclear, since we are no experts in it. This does not permit us to judge whether mCRL2 performs better or worse in terms of adaptability.

A similar reasoning holds for Statecharts. The authors describe in a fairly easy way how to obtain a simple switch from the original switch. However, the subsequent discussion they show that the presented model of the original

switch is incorrect and requires a more complicated model to capture the design intent. Since this correct complicated model is not given, it is not fair to make a comparison.

For modeling the modified switch in Bluespec, the authors require an entire redesign of the original switch, such that each priority separately defines rules. This leads to almost a duplication of the model. As we compare the same extension for our modified switch in mCRL2, we only have to split a summand and alter a guard, which is a rather small modification.

ACP serves well in terms of adaptability for this case study. As mCRL2 falls in the same category as ACP, this also holds for mCRL2. Therefore in terms of adaptability, mCRL2 and ACP are comparable.

Furthermore, we have set up the processes of the buffers in such a way that they can be easily reused for a more general specification, e.g. a $N \times M$ specification. To do so, we are required to add extra process references in the initialisation, and add extra rules to the data equations for routing packets. Within the current models we allow, that only one packet can be send simultaneously within a clock cycle. By adding more processes, this bound will not change. To increase the throughput, e.g. allowing more message transfers per clock cycle, we need to add summands that grant this communication. We argue that these modifications can be done at a local level.

7.4 Verification

The authors of [6] are unable to convince themselves that the specification they give are correct with respect to the design intent. As their remark essentially holds for all specifications, it already shows the first pitfalls in concurrent system design.

In line with the authors of [6], we agree that global reasoning is required on a specification across all the processes to verify system requirements. This however is a difficult task. As the description of the models is fairly simple, their explicit behaviour is not. In Figure 1 we have taken the opportunity to show, that even for a small system like the simplified switch, it already leads to systems that cannot be overlooked by human reasoning¹. For a buffer capacity of three elements, we generate a state-space of 3600 states with 41137 transitions². Nevertheless, with the automated methods of the mCRL2 tool set it turns out to be possible to verify interesting properties of the modelled systems. This does not require reasoning by humans, which is the case for establishing properties using the formalisms used in [6].

¹ These numbers are obtained, without applying any reduction techniques. We are aware that these numbers can be reduced. Note that the number of states and the number of transitions are given on a logarithmic scale.

² Please note that multi-actions that contain precisely the same actions are only taken into account once. Otherwise, the numbers of transitions would have been much larger.

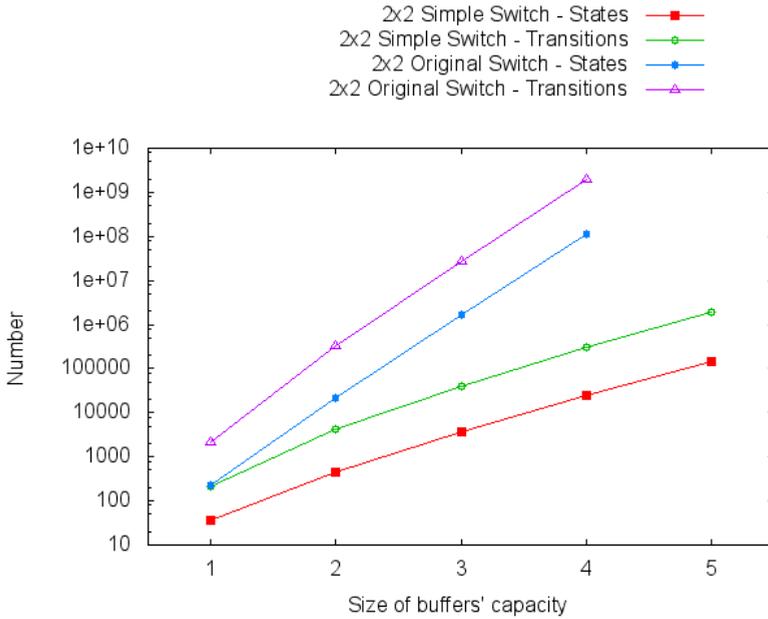


Fig. 1. Complexity of the model expressed in number of states and transitions for the simplified and original switch models

8 Conclusion and Future Work

In this paper, we have shown, in a case-study, that mCRL2 is suitable for the modelling and subsequent analysis of a system in which multi-party communications combined with priority-based communication occur. We have tried to apply local reasoning as much as possible, by generalizing the behavior of the buffers by type, thereby preserving both the possibility to send prioritized packets as well as sending packets simultaneously. As a consequence, it is possible to re-use the models in a more general setting. Furthermore, we showed that with mCRL2, we were also able to verify some properties, that has led to an increase in confidence that the model represents the design intent. Thereby, we have shown that mCRL2 is at least comparable to the formalisms used in [6], and in some cases more suitable for specifying complex system designs.

We should note that the comparison is based on subjective grounds. For a fair comparison, one should study the possible language constructs for each of the formalisms and point out the differences. This requires an expert over multiple formalisms or a cooperation among experts of different formalisms. Since the case study is centered around a specific specification, for which the models are created according to the level of expertise of the designers, the outcome of the comparison is subjective. As the authors of this paper can be considered experts when it comes to mCRL2 specifications, and are familiar with ACP and Statecharts, we are confident about the claims made between these formalisms.

We have shown that it is possible to capture relative performance requirements, without explicitly stating time. Since mCRL2 falls into the category of timed process algebra's [2], it allows designers to specify real-time behaviour. Nevertheless, we have chosen not to do so for several reasons. First, we would like to have a fair comparison between the untimed formalisms. Second, timed requirements tend to be complex in general and require challenging manipulations on the mCRL2 models before one can verify requirements. Nevertheless, we believe that the case study considered in this paper can be formulated in a timed specification, and can serve as subject of study for reduction and analysis techniques for timed systems.

References

1. Baeten, J.C.M., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. Cambridge tracts in theoretical computer science, vol. 50. Cambridge University Press, Cambridge (2010)
2. Baeten, J.C.M., Kees Middelburg, C.A.: Process Algebra with Timing. In: EATCS Monographs, Springer, Berlin (2002)
3. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. *Information and Control* 60(1-3), 109–137 (1984)
4. Bluespec: Automatic Generation of Control Logic with Bluespec SystemVerilog (Februari 2005), <http://www.bluespec.com/forum/download.php?id=63>
5. Bradfield, J.C.: Verifying Temporal Properties of Systems. In: Progress in Theoretical Computer Science, Birkhäuser, Basel (1992)
6. Daylight, E.G., Shukla, S.K.: On the difficulties of concurrent-system design, illustrated with a 2×2 switch case study. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 273–288. Springer, Heidelberg (2009)
7. Groote, J.F., Keiren, J., Mathijssen, A., Ploeger, B., Stappers, F., Tankink, C., Usenko, Y., van Weerdenburg, M., Wesselink, W., Willemse, T., van der Wulp, J.: The mCRL2 toolset. In: Proceedings International Workshop on Advanced Software Development Tools and Techniques, WASDeTT 2008 (2008)
8. Groote, J.F., Mateescu, R.: Verification of temporal properties of processes in a setting with data. In: Haeberer, A.M. (ed.) AMAST 1998. LNCS, vol. 1548, pp. 74–90. Springer, Heidelberg (1998)
9. Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., van Weerdenburg, M.: The formal specification language mCRL2. In: Brinksma, E., Harel, D., Mader, A., Stevens, P., Wieringa, R. (eds.) Methods for Modelling Software Systems (MMOSS). Dagstuhl Seminar Proceedings, vol. 06351, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007)
10. Groote, J.F., Mathijssen, A.H.J., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: Analysis of distributed systems with mCRL2. In: Alexander, M., Gardner, W. (eds.) Process Algebra for Parallel and Distributed Processing, ch. 4, pp. 99–128. Taylor & Francis, Abington (2009)
11. Harel, D.: Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.* 8(3), 231–274 (1987)

12. Hoe, J.C., Arvind: Synthesis of operation-centric hardware descriptions. In: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design, ICCAD 2000, Piscataway, NJ, USA, pp. 511–519. IEEE Press, Los Alamitos (2000)
13. Kozen, D.: Results on the propositional mu-calculus. *Theor. Comput. Sci.* 27, 333–354 (1983)
14. Lamport, L.: *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Reading (2002)
15. Singh, G., Shukla, S.K.: Verifying compiler based refinement of BluespecTM specifications using the spin model checker. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) *SPIN 2008*. LNCS, vol. 5156, pp. 250–269. Springer, Heidelberg (2008)