Sergey Ostroumov | Marina Waldén

Formal Library of Visual Components

Turku Centre *for* Computer Science

TUCS Technical Report
No 1147, November 2015

Formal Library of Visual Components

Sergey Ostroumov
Åbo Akademi University, Faculty of Science and Engineering
Joukahaisenkatu 3-5A, 20520, Turku, Finland
Sergey.Ostroumov@abo.fi

Marina Waldén
Åbo Akademi University, Faculty of Science and Engineering
Joukahaisenkatu 3-5A, 20520, Turku, Finland
Marina.Walden@abo.fi

TUCS Technical Report
No 1147, November 2015

Abstract

Scalability and reusability are the limiting factors in using formal methods such as Event-B in complex system development. In addition, the formal Event-B specification of a system requires background knowledge, which prevents a fruitful communication between the developer and the customer. On the other hand, the formal development in Event-B provides a means for system-level specification and verification supported by the correctness proof. The development in Event-B follows the refinement approach, in which the specification is created top-down starting from a non-deterministic model and ending in the precise implementable one. The specification process is supported by theorem proving, so that one can guarantee correctness of the specification with respect to postulated properties called invariants. Event-B also has a mature tool support, namely the Rodin platform, which generates and attempts to automatically discharge the necessary proof obligations.

This paper presents an approach which is to facilitate scalability of formal development in Event-B. We aim to build a formal library of parameterized visual components that can be reused whenever needed. Each component is formally developed and proved correct by utilizing the advantages of Event-B. Furthermore, each component has a unique graphical representation which eases the rigorous development process by applying the "drag-and-drop" approach and enhances the communication between a developer and a customer. We present a subset of components from different application domains.

Keywords: Components Library, Event-B, Formal Components, Human-Machine Interface, Visual Design

TUCS Laboratory
RITES – Resilient IT Infrastructures
Distributed Systems Laboratory
Integrated Design of Quality Systems group

The work was done within the Advices project funded by Academy of Finland, grant No. 266373.

1. Introduction

Event-B [1] is a formal method that allows designers to build systems in such a manner that the correctness of the development process is supported by mathematical proofs. The development process proceeds in a top-down fashion starting from an abstract (usually non-deterministic) specification. This specification is then stepwise refined by adding the details about the system until the implementable level is reached. The process of transforming an abstract specification into an implementable one via a number of correctness preserving steps is known as refinement [2]. This mechanism allows the developers to build systems in a stepwise and correct-by-construction manner.

The specification (or the model) of a system in Event-B captures the functional behaviour as well as the essential properties that must hold (invariants). The refinement approach helps the designers to deal with the system requirements in a stepwise manner, which makes the correctness proof along the development easier. However, as more details are added to the system specification, it becomes complex and hard to handle. This limits the scalability of this approach. Moreover, the more details are present in the specification, the harder it is to convince the stake holders about the fact that the system specification embodies all the necessary requirements.

To facilitate scalability of formal development in Event-B and enhance communication between the developer and stakeholder, we aim to build a formal library of parameterized visual components. Each component is formally developed and proved correct by utilizing the Event-B engine. Moreover, each component is tied to a unique graphical representation. The development process then proceeds according to the “drag-and-drop” approach, where the developer picks the necessary components from the library and instantiates them. Since the components are parameterized and are in the library, they can be reused in various application domains depending on the requirements. The specification of a system is then a visual model whose correctness is supported by the underlying Event-B language. Visual design eases the rigorous development process and facilitates the communication between a developer and a customer. We present a pattern for the development of components and illustrate a subset of formal components from different application domains developed in the proposed manner.

The remainder of the paper is organized as follows. The next section gives an overview on the existing approaches related to the rigorous component-based design. Section 3 describes the notation of Event-B and proof obligations that provide the correctness proof. Section 4 presents a subset of parameterized visual components that have been formally developed and added to the library. Finally, Section 5 concludes the paper the outlines the directions of the future work.

2. Related Work

BMotionStudio has been proposed as an approach to visual simulation of the Event-B models [3][4]. The idea behind BMotionStudio is that the designer creates a domain specific image and links the model to it using a gluing code written in JavaScripts. The simulation is based on ProB animator and model checker [5], so that whenever the model is executed the corresponding graphical element reacts on the changes. The BMotionStudio tool also supports interaction with the user, i.e., a user can provide an input through visual elements instead of manipulating the model directly.

In contrast to the BMotionStudio approach, we aim for creating a formal library of the predefined components that already have the visual representation. The development of the specification is then a process of the instantiation of the necessary components and the connection of them into a system. That is, the developer does not need to redraw the graphical representation, but to simply reuse the components.

Eventually, the designer obtains a graphical representation of the system whereas its specification is in fact written in Event-B and supported by the correctness proof. Certainly, our approach can be complemented by the BMotionStudio in order to obtain visualisation of the model execution.

Snook and Butler [6] have proposed an approach to merge visual UML [7] that lacks formal precise semantics with B [8] that requires significant effort in training to overcome the mathematical barrier. This approach has then been extended to Event-B and called iUML-B [9]. The authors define semantics of UML by translating it to Event-B. The use of UML-B profile provides specialisation of UML entities to support refinement. The authors also present the tools that automatically generate an Event-B model from UML.

In contrast to using UML as visualisation tool, we aim to create a formal library of parameterised components, each of which has its own graphical representation. The system specification is then a visual model that represents a composition of the instantiated versions of these components. Nevertheless, we target automated generation of the necessary data structures and Event-B elements whenever our approach is applied.

An approach to a component-based formal design within Event-B has been proposed by Ostroumov, Tsiopoulos, Plosila and Sere [10]. The aim of this work is the generation of a structural VHDL [11] description from a formal Event-B model. The authors present a one-to-one mapping between formal functions defined in an Event-B context and library components derived from VHDL. Using this mapping, the authors rely on an additional refinement step, in which regular operations are replaced with function calls. This allows for automated generation of structural VHDL descriptions.

In contrast to this approach, we propose an approach to systems development in Event-B in a visual manner. This approach is not limited to VHDL descriptions and allows the designers to utilize various components from different application domains. Moreover, we aim to create a formal library parameterized Event-B specifications which capture the generic behaviour of these components. This approach is to facilitate the reuse of the components, where the developers can develop the system in a “drag-and-drop” manner.

3. Preliminaries: Event-B

The Event-B formalism [1] offers several advantages. First, it allows us to build system level models. Second, it supports the refinement approach such that a model is built top-down in a correct-by-construction manner. Third, the development follows rigorous rules with mathematical proofs of correctness of models. Last but not least, it has a mature tool support extensible in the form of plug-ins, namely the Rodin platform [13]. Let us now describe the structure and notation of Event-B (Fig. 1).

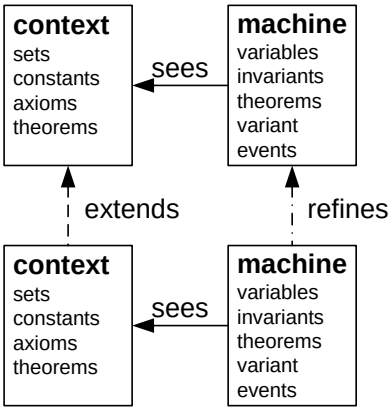


Figure 1. Event-B contexts and machines: contents and relationship [1]

3.1. Event-B Model Structure

A specification in Event-B consists of *contexts* and *machines*. The relationship between them is as shown in Fig. 1. A context can be *extended* by another context whilst a machine can be *refined* by another machine. Moreover, a machine can refer to the contents of the context (to "*see*").

A context specifies static structures such as data types in terms of *sets*, *constants*, properties given as a set of *axioms*. One can also postulate and prove *theorems* that ease proving effort during the model development.

A machine models the behaviour of a system. The machine includes *state variables*, *theorems*, *invariants*, a *variant* and guarded transitions (*events*). The invariants represent constraining predicates that define types of the state variables as well as essential properties of the system. The overall system invariant is defined as the conjunction of these predicates.

A variant is a natural number or a finite set. It is required to show the termination of certain events that can be executed several times in a row, e.g., modelling a loop.

An event describes a transition from a state to a state. The syntax of the event is as follows:

$$E = \textbf{ANY}\ x\ \textbf{WHERE}\ g\ \textbf{THEN}\ a\ \textbf{END}$$

where x is a list of event local variables. The *guard* g stands for a conjunction of predicates over the state variables and the local variables. The *action* a describes a collection of assignments to the state variables.

We can observe that an event models a guarded transition. When the guard g holds, the transition can take place. In case several guards hold simultaneously, any of the enabled transitions can be chosen for execution non-deterministically. If none of the guards holds, there is a deadlock.

When a transition takes place, the action a is performed. The action a is a composition of the assignments to the state variables executed simultaneously and denoted as $||$. An assignment can be either deterministic or non-deterministic. A deterministic assignment is defined as $v := E(w)$, where v is a list of state variables, E is a list of expressions over some set of state variables w. A non-deterministic assignment is specified as $v :| Q(w, v')$, where $Q(w, v')$ is a predicate over some state variables w and a new value v' of variable v. The variable v obtains such a value v' that $Q(w, v')$ holds.

3.2. Event-B Proof Mechanism

These denotations allow for describing semantics of Event-B in terms of *before-after predicates* (BA) [14]. Essentially, a transition is a BA that establishes a relationship between the model state before (v) and after (v') the execution of an event. Hence, the correctness of the model is verified by checking if the events preserve the invariants (INV) and are feasible to execute (FIS) in case the event action is non-deterministic:

$$Inv \wedge g_e \Rightarrow [BA_e]Inv \qquad \text{(INV)}$$

$$Inv \wedge g_e \Rightarrow \exists v' \,.\, BA_e \qquad \text{(FIS)}$$

where Inv is a model invariant, g_e and BA_e are the guard and the before-after predicate of the event e, respectively. The expression $[BA_e]Inv$ stands for the substitution in the invariant Inv according to BA_e.

In addition, deadlock freedom of the specification may be corroborated. A deadlock free specification stands for the case where there exists at least one event that can be executed. To achieve this, one needs to postulate a machine theorem that includes the guards of all the events connected with disjunction and show that the proof obligation (DLF) [1] is preserved:

$$\forall S, C, V \,.\, A \wedge I \Rightarrow \bigvee_{i=1}^{n} g_i \qquad \text{(DLF)}$$

where n is the number of events and g_i is the guard of the i-th event. The structures S, C and A represent sets, a collection of constants and axioms introduced into a context, respectively. The structures V and I stand for a set of state variables and a set of invariants of a machine, respectively.

3.3. Refinement in Event-B

Since the specification development in Event-B follows the refinement approach, one has to prove that the more concrete (refined) events simulate their abstract counterparts. To show this, the refined events must preserve the guard strengthening (GRD) and action simulation (SIM) proof obligations [15] as well:

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \,.\, A \wedge A_r \wedge I \wedge I_r \wedge g_r \Rightarrow g \qquad \text{(GRD)}$$

$$\forall S, C, S_r, C_r, V, V_r, x, x_r \,.\, A \wedge A_r \wedge I \wedge I_r \wedge BA_{er} \Rightarrow BA_e \qquad \text{(SIM)}$$

where all letters with subscript "r" stand for the refined versions of the aforementioned structures.

To prove that new events executed several times in a row terminate, one also has to show that these events are consistent with a variant. In particular, these events have to preserve either of the following proof obligations depending on whether the variant is a natural number (VAR_N) or a finite set (VAR_S) [15]:

$$\forall S, C, V \,.\, A \wedge I \Rightarrow Var \in \mathbb{N} \wedge [BA_e]Var < Var \qquad \text{(VAR\_N)}$$

$$\forall S, C, V \,.\, A \wedge I \Rightarrow finite(Var) \wedge card([BA_e]Var) < card(Var) \qquad \text{(VAR\_S)}$$

where Var is a variant that denotes a numeric expression or a finite set of values. The expressions finite(Var) and card(Var) specify finitness and cardinality of the set variant, respectively.

In case the model needs to be deadlock free, one can show the relative deadlock freedom, i.e., all concrete events should not deadlock more frequently than the abstract ones. Therefore, the disjunction of the abstract guards should imply the disjunction of the concrete guards (proof obligation (DLFR)) [1]:

$$\forall S, C, V \,.\, A \wedge I \wedge I_r \wedge \bigvee_{i=1}^{n} g_i \Rightarrow \bigvee_{j=1}^{m} g_{rj} \qquad \text{(DLFR)}$$

where m is the number of concrete events and g_j is the guard of the j-th event.

The Rodin platform [13], a tool support for Event-B, automatically generates and attempts to discharge (prove) the necessary proof obligations. The best practices encompass the development of the specification in such a manner that 90-95% of the proof obligations are discharged automatically. However, the tool sometimes requires the user assistance provided via the interactive prover. Typically, the claims that are difficult for the automatic prover to discharge require case distinction and/or data substitution.

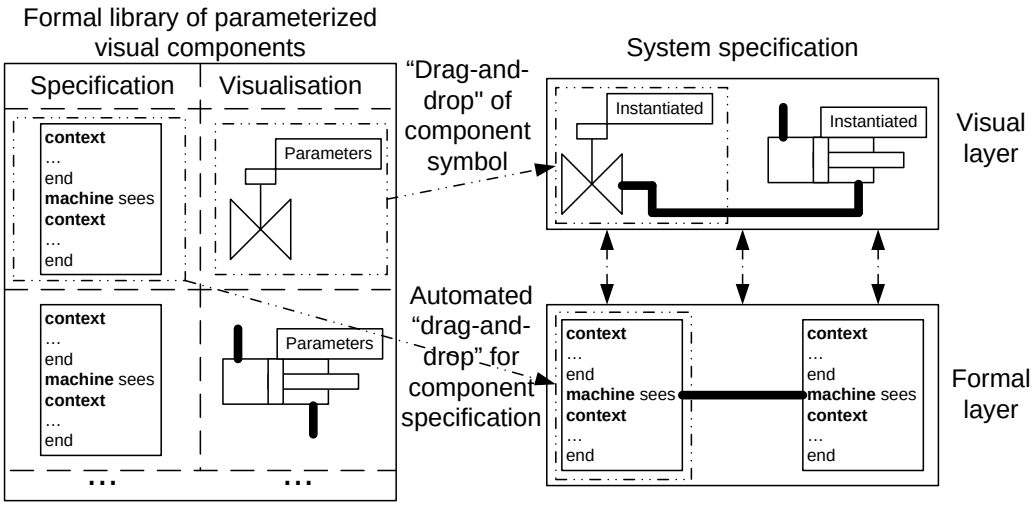


Figure 2. "Drag-and-drop" approach for visual system desing in Event-B

4. Library of Formal Components

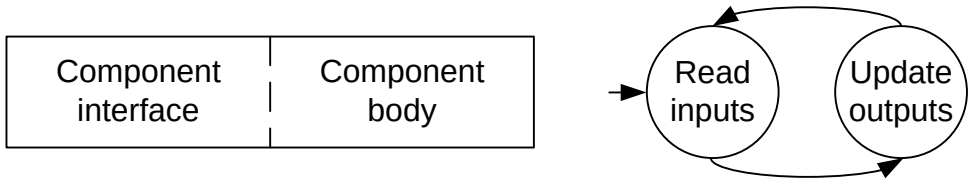
We aim to create a formal library of components. Each component is developed formally within the Event-B formal framework and is tied to a unique graphical symbol. Moreover, the components in the library have to be parameterized whenever possible in order to be reusable during the development process (Fig. 2).

The system specification/development is then a process of picking the needed components, instantiating and connecting them. That is, it is the process, where the system is developed out of components in the "drag-and-drop" fashion (see Fig. 2). Notice that the connectivity between the components is a separate problem which is out of the scope of this paper. The focus of this paper is on the formal library of visual components, which is to facilitate reusability and scalability of the rigorous formal development in such formalisms as Event-B.

In the following sub-sections, we overview several components from different application domains. Particularly, we present two components from (digital) hydraulics domain, namely an electro-valve and a cylinder, as well as two components from railway domain, namely a point and a crossing.

4.1. Component Functionality

Let us start by describing the generic functionality of a component. A component is a reactive device that updates its outputs according to the input stimulate. The component typically consists of two parts: an interface and a body (Fig. 3, a). The interface is comprised of the set of inputs and outputs that are seen by the outside world whilst the body performs the component functions. Generally, the operation of the component has to be deterministic. That is, the same input stimuli must generate the same output results and the order of operations to compute these outputs according to the input stimuli is known a priori. To achieve this, we use a common pattern for control systems [12], in which the component first reads the inputs (environment) and then produces the outputs (control). In other words, a component has at least two alternating modes: read of the inputs and production of the outputs (Fig. 3, b)). These operations repeat indefinitely. Therefore, the main property of a component is that it must always work (i.e., it never terminates – deadlock free).



a) b)

Figure 3. A component pattern: a) structure of a component, b) automaton

4.2. Hydraulic components: an electro-valve

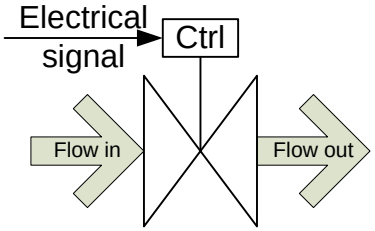


Figure 4. A symbolic representation of an electro-valve with the interface

The first example of a visual component is an electro-valve (Fig. 4). The electro-valve is a physical device that transfers a flow of a liquid from one port to another and is controlled by an electrical signal. The application of a positive control signal opens the valve whilst the negative signal closes it. If no signal is present on the control input, the valve keeps the current position. Moreover, the valve opens and closes with some rate due to the physical laws.

The complete Event-B model of an electro-valve can be found in Appendix A. This model of a valve has the following parameters: the minimum (valve_diameter_min_val) and the maximum (valve_diameter_max_val) flow the valve can let trough and the rate (valve_rate) with which the valve opens and closes. The rate cannot be greater than the difference between the maximum and the minimum flow. Assuming that when the valve is closed the outlet is fully closed as well (i.e., no flow can come through), the minimum flow equals to zero and the rate cannot be greater than the maximum diameter of the valve. Moreover, if the rate equals to the maximum, the valve is simply open/close. The minimum flow, the maximum flow and the rate parameters as well as the set of control signals are captured by the Event-B constants introduced into a context as shown in Listing 1.

Listing 1. The parameters of a generic valve

```
context Valve_parameters
constants
  valve_diameter_min_val
  valve_diameter_max_val
  valve_CONTROL
  valve_rate
axioms
  valve_diameter_min_val = 0 ∧ valve_diameter_max_val ∈ ℕ1 ∧ valve_CONTROL = {−1,0,1}  ∧
  valve_rate ∈ ℕ1  ∧ valve_rate ≤ valve_diameter_max_val − valve_diameter_min_val
end
```

The interface of a valve consists of two inputs and one output, namely the control signal, the input port and the output port, respectively. Additionally, the valve has a variable that shows the current position of the plunger in the valve and the mode variable that models the transitions between the inputs read and outputs production states. The variables constitute the state space of the valve model (Listing 2).

Listing 2. The state space of a valve

```
machine Valve_Behaviour sees Valve_parameters
variables
  valve_control_I
  valve_flow_I
  valve_flow_O
  valve_mode
  valve_position
invariants
  valve_control_I ∈ valve_CONTROL ∧ valve_mode ∈ 0..1 ∧
  valve_flow_I ∈ valve_diameter_min_val..valve_diameter_max_val ∧
  valve_flow_O ∈ valve_diameter_min_val..valve_diameter_max_val ∧
  valve_position ∈ valve_diameter_min_val..valve_diameter_max_val
```

The main property of the valve is that the flow from the output port cannot be greater than the flow on the input port (valve_mode = 0 ⇒ valve_flow_O ≤ valve_flow_I). Moreover, the position of the plunge regulates the output flow, so that the output flow cannot be stronger than allowed (valve_flow_O ≤ valve_position). Additionally, the output flow has to be always updated when the new inputs are read (i.e., the non-termination property as it was stated earlier). The former properties are captured as invariants. The latter is stated as a deadlock freedom theorem (Listing 3) which is the disjunction of the guards of the component events. The theorem evaluates to true, which supports the fact that the component always works.

Listing 3. The properties of a valve

```
// The output flow cannot be stronger than the input flow and cannot be larger than the opening of the valve
(valve_mode = 0 ⇒ valve_flow_O ≤ valve_flow_I ∧ valve_flow_O ≤ valve_position) ∧
// The property of never terminating
```

```
theorem (valve_mode = 0 ∨
        (valve_control_I = 1 ∧ valve_position + valve_rate ≤ valve_diameter_max_val ∧ valve_mode = 1) ∨
        (valve_control_I = −1 ∧ valve_position − valve_rate ≥ valve_diameter_min_val ∧ valve_mode = 1) ∨
        ((valve_control_I = 0 ∨ (valve_position − valve_rate < valve_diameter_min_val ∧ valve_control_I = −1) ∨
         (valve_position + valve_rate > valve_diameter_max_val ∧ valve_control_I = 1)) ∧ valve_mode = 1))
```

The functionality of the valve includes: reading the control signal and the input flow, opening the valve, closing the valve and keeping the previous position (i.e., neither opening nor closing). Initially, the valve is idle, there might be some input flow, but the valve is closed, hence, there is no output flow. The mode is set to reading the new inputs (Listing 4).

Listing 4. The initialization of a valve

```
event INITIALISATION
 then
  valve_control_I ≔ 0 || valve_mode ≔ 0 || valve_flow_I :∈ valve_diameter_min_val..valve_diameter_max_val ||
  valve_flow_O ≔ valve_diameter_min_val || valve_position ≔ valve_diameter_min_val
end
```

In order for a valve to produce the intended outputs, the valve first needs to read the inputs. This is usually considered as an environmental event that updates the inputs of the model. We assume that all inputs of the valve are updated simultaneously as shown in Listing 5. Notice that the input read is specified as a non-deterministic operation bounded to the parameters of the valve.

Listing 5. The read of the inputs

```
event valve_environment
 where
  valve_mode = 0
 then
  valve_mode ≔ 1 || valve_control_I :∈ valve_CONTROL ||
  valve_flow_I :∈ valve_diameter_min_val..valve_diameter_max_val
end
```

Once the inputs are read (valve_mode = 1), the valve can perform the following operations: open with some rate, close with the same rate or keep the current position. These operations are modelled using the three corresponding events shown below.

The valve opening event (Listing 6) can clearly take place when the control signal (the command) is to open the valve (valve_control_I = 1). However, the valve cannot open more than allowed, that is, it cannot be open more than the maximum diameter of the valve (valve_position + valve_rate ≤ valve_diameter_max_val). When the valve is opening, the output flow increases according to the input flow and the current position of the plunge (valve_position + valve_rate < valve_flow_I ⇒ *valve_flow_O_new* = valve_position + valve_rate). Notice however that if the diameter of the valve allows stronger than the input flow to come through, the output flow is simply the same as the input one (valve_position + valve_rate ≥ valve_flow_I ⇒ *valve_flow_O_new* = valve_flow_I).

Listing 6. The stepwise opening of a valve

```
event valve_opening
 any valve_flow_O_new
 where
  valve_control_I = 1 ∧ valve_mode = 1 ∧ (valve_position + valve_rate ≤ valve_diameter_max_val) ∧
  (valve_position + valve_rate < valve_flow_I ⇒ valve_flow_O_new = valve_position + valve_rate) ∧
  (valve_position + valve_rate ≥ valve_flow_I ⇒ valve_flow_O_new = valve_flow_I)
 then
  valve_flow_O ≔ valve_flow_O_new || valve_mode ≔ 0 || valve_position ≔ valve_position + valve_rate
end
```

The valve closing event is evidently opposite to the opening of the valve (Listing 7). It can take place when the command is to close the valve (valve_control_I = −1). When the valve is closing, the output flow decreases with some rate (valve_position − **valve_rate** ≤ valve_flow_I ⇒ *valve_flow_O_new* = valve_position − **valve_rate**). Notice that the valve can still allow for stronger than the input flow after a step of closing, in which case the output remains the same as the input flow (valve_position − **valve_rate** > valve_flow_I ⇒ *valve_flow_O_new* = valve_flow_I). Notice also that the valve can close with some rate if it is not completely closed yet (valve_position − **valve_rate** ≥ **valve_diameter_min_val**).

Listing 7. The stepwise closing of a valve

```
event valve_closing
 any valve_flow_O_new
 where
  valve_control_I = −1 ∧ valve_mode = 1 ∧ valve_position − valve_rate ≥ valve_diameter_min_val ∧
  (valve_position − valve_rate ≤ valve_flow_I ⇒ valve_flow_O_new = valve_position − valve_rate) ∧
  (valve_position − valve_rate > valve_flow_I ⇒ valve_flow_O_new = valve_flow_I)
 then
  valve_flow_O ≔ valve_flow_O_new || valve_mode ≔ 0 || valve_position ≔ valve_position − valve_rate
end
```

Finally, if the command is neither open nor close (valve_control_I = 0) or the plunge has reached its minimum (valve_position − **valve_rate** < **valve_diameter_min_val** ∧ valve_control_I = −1) or maximum (valve_position + **valve_rate** > **valve_diameter_max_val** ∧ valve_control_I = 1), the valve keeps the current position of the plunge. In other words, the valve is idle or stopped. Therefore, the output flow remains unchanged with respect to the current flow (valve_flow_I ≥ valve_flow_O ⇒ *valve_flow_O_new* = valve_flow_O) and the input flow (valve_flow_I < valve_flow_O ⇒ *valve_flow_O_new* = valve_flow_I). Listing 8 summarizes this functionality.

Listing 8. The idle state of a valve

```
event valve_stop // The valve is stopped in all other cases not captured by the aforementioned events
 any valve_flow_O_new
 where
   valve_mode = 1 ∧
  (valve_control_I = 0 ∨ (valve_position − valve_rate < valve_diameter_min_val ∧ valve_control_I = −1) ∨
    (valve_position + valve_rate > valve_diameter_max_val ∧ valve_control_I = 1)) ∧
  (valve_flow_I < valve_flow_O ⇒ valve_flow_O_new = valve_flow_I) ∧
  (valve_flow_I ≥ valve_flow_O ⇒ valve_flow_O_new = valve_flow_O)
 then
  valve_mode ≔ 0 || valve_flow_O ≔ valve_flow_O_new
end
```

4.3. Hydraulic components: a cylinder

A cylinder is a pure hydraulic component in comparison with an electro-valve. That is, the cylinder reacts on liquid flows only and does not have any electrical inputs. Nonetheless, it is a reactive device which updates its output according to the input stimulate. The visual symbol of a cylinder is shown in Fig. 5.

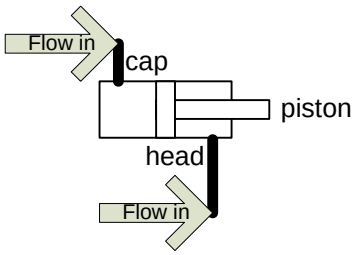


Figure 5. Visual represeantion of a cylinder

A cylinder has two inputs (cap and head) and one output. The inputs allow the liquid to flow into the body of the cylinder. The output of the cylinder is the piston that moves according to the difference in the input flows. The piston moves forward if the difference between the liquid flow coming into the cap and the liquid flow coming into the head is positive. The piston moves backward if this difference is negative. Clearly, if the difference equals to 0 (i.e., the flows are the same or there are no flows), the piston keeps the position. Due to physical laws, the piston moves with some rate. This rate is determined by the difference in the input flows as well.

The complete formal model of a cylinder can be found in Appendix B. The cylinder model has the four parameters (Listing 9). Two of them (cylinder_input_diameter_min_val and cylinder_input_diameter_max_val) define the diameters of the inputs. That is, the possible liquid flows that can be applied to the cylinder. We assume that both inputs are of the same size, so that the motion of the piston is proper. The other two specify (cylinder_head_pos and cylinder_cap_pos) the limits of the piston motion. In other words, the difference between cylinder_head_pos and cylinder_cap_pos determines the length along which the piston can move.

Listing 9. Parameters of a cylinder

```
context Cylinder_parameters
constants
  cylinder_input_diameter_min_val
  cylinder_input_diameter_max_val
  cylinder_cap_pos
  cylinder_head_pos
axioms
  cylinder_input_diameter_min_val = 0 ∧ cylinder_input_diameter_max_val ∈ ℕ1 ∧
  cylinder_cap_pos = 0 ∧ cylinder_head_pos ∈ ℕ1
end
```

As mentioned above, a cylinder has two inputs, namely cylinder_flow_cap_I and cylinder_flow_head_I, as well as one output, namely cylinder_piston_position_O. There is also a variable that specifies the modes of the cylinder component, namely cylinder_mode (Listing 10). The cylinder does not have special properties except for the non-termination. This property is captured by the deadlock freedom theorem. The theorem evaluates to true, which supports the fact that the cylinder always operates.

Listing 10. Cylinder variables and properties

```
machine Cylinder_behaviour sees Cylinder_parameters
variables
  cylinder_flow_cap_I
  cylinder_flow_head_I
  cylinder_piston_position_O
  cylinder_mode
invariants
  // Current position of the piston in the cylinder
  cylinder_piston_position_O ∈ cylinder_cap_pos..cylinder_head_pos ∧
  // Input to move the piston to the right
  cylinder_flow_cap_I ∈ cylinder_input_diameter_min_val..cylinder_input_diameter_max_val ∧
  // Input to move the piston to the left
  cylinder_flow_head_I ∈ cylinder_input_diameter_min_val..cylinder_input_diameter_max_val ∧
  cylinder_mode ∈ 0..1 ∧
  // Deadlock freedom – non-termination
  theorem cylinder_mode = 0 ∨ (cylinder_flow_cap_I > cylinder_input_diameter_min_val ∧
                cylinder_flow_cap_I > cylinder_flow_head_I ∧
                cylinder_piston_position_O + cylinder_flow_cap_I − cylinder_flow_head_I ≤ cylinder_head_pos ∧
                cylinder_mode = 1) ∨
               (cylinder_flow_head_I > cylinder_input_diameter_min_val ∧
                cylinder_flow_head_I > cylinder_flow_cap_I ∧
```

```
cylinder_cap_pos ≤ cylinder_piston_position_O + cylinder_flow_cap_I − cylinder_flow_head_I ∧
cylinder_mode = 1) ∨
(∃cylinder_rate . cylinder_rate = cylinder_flow_cap_I − cylinder_flow_head_I ∧ cylinder_mode = 1 ∧
(cylinder_flow_head_I = cylinder_flow_cap_I ∨
 cylinder_piston_position_O + cylinder_rate < cylinder_cap_pos ∨
 cylinder_piston_position_O + cylinder_rate > cylinder_head_pos))
```

Initially, there are no input flows, the piston is at some position within the cylinder body and the mode is set to reading the inputs. In order for the piston to move, both of the inputs have to be updated. This is achieved by the environment event shown in Listing 11.

Listing 11. Update of the cylinder inputs

```
event cylinder_environment
 where
  cylinder_mode = 0
 then
  cylinder_flow_cap_I :∈ cylinder_input_diameter_min_val..cylinder_input_diameter_max_val ||
  cylinder_flow_head_I :∈ cylinder_input_diameter_min_val..cylinder_input_diameter_max_val ||
  cylinder_mode ≔ 1
end
```

As mentioned above, there are three possible reactions to the input flows. First, the piston can move forward (extend), if the flow coming into cap is larger than the flow coming into head (cylinder_flow_cap_I > cylinder_flow_head_I). Indeed, the flow must be present on the cap input (cylinder_flow_cap_I > cylinder_input_diameter_min_val) and there has to be space for the piston to extend (cylinder_piston_position_O + *cylinder_rate* ≤ cylinder_head_pos). If these conditions are met, the piston extends with a rate which equals to the difference between the input flows (Listing 12).

Listing 12. Motion of the piston forward (extend)

```
event cylinder_extending
 any cylinder_rate
 where
  cylinder_rate = cylinder_flow_cap_I − cylinder_flow_head_I ∧ cylinder_mode = 1 ∧
  cylinder_flow_cap_I > cylinder_flow_head_I ∧ cylinder_flow_cap_I > cylinder_input_diameter_min_val ∧
  cylinder_piston_position_O + cylinder_rate ≤ cylinder_head_pos
 then
  cylinder_mode ≔ 0 || cylinder_piston_position_O ≔ cylinder_piston_position_O + cylinder_rate
end
```

Second, the piston can retract. This can occur if the flow on the head input is larger than on the cap input (cylinder_flow_head_I > cylinder_flow_cap_I). Moreover, the flow on the head has to be present (cylinder_flow_head_I > cylinder_input_diameter_min_val) and there is space for the piston to retract (cylinder_cap_pos ≤ cylinder_piston_position_O + *cylinder_rate*). Listing 13 captures this behavior.

Listing 13. Motion of the piston backward (retract)

```
event cylinder_retracting
 any cylinder_rate
 where
  cylinder_rate = cylinder_flow_cap_I − cylinder_flow_head_I ∧ cylinder_mode = 1 ∧
  cylinder_flow_head_I > cylinder_flow_cap_I ∧ cylinder_flow_head_I > cylinder_input_diameter_min_val ∧
  cylinder_cap_pos ≤ cylinder_piston_position_O + cylinder_rate
 then
  cylinder_mode ≔ 0 || cylinder_piston_position_O ≔ cylinder_piston_position_O + cylinder_rate
end
```

Finally, if the flows are the same (cylinder_flow_head_I = cylinder_flow_cap_I) or there is no space for the piston to extend (cylinder_piston_position_O + *cylinder_rate* > **cylinder_head_pos**) nor to retract (cylinder_piston_position_O + *cylinder_rate* < **cylinder_cap_pos**), the piston keeps its position. In other words, the piston is stopped (Listing 14).

Listing 14. Keep the position of the piston (stop)

```
event cylinder_stop
 any cylinder_rate
 where
  cylinder_rate = cylinder_flow_cap_I − cylinder_flow_head_I ∧ cylinder_mode = 1 ∧
  (cylinder_flow_head_I = cylinder_flow_cap_I ∨
   cylinder_piston_position_O + cylinder_rate > cylinder_head_pos ∨
   cylinder_piston_position_O + cylinder_rate < cylinder_cap_pos)
 then
  cylinder_mode ≔ 0
end
```

4.4. Railway components: a point

Let us now review a couple of components developed in the proposed manner, but from a different application domain, namely from the railway domain. The components we show here have bidirectional ports, i.e., each port of the component can be read as the input and updated as the output. We start by presenting a point – a type of a railway comprising three ways (Fig. 6).

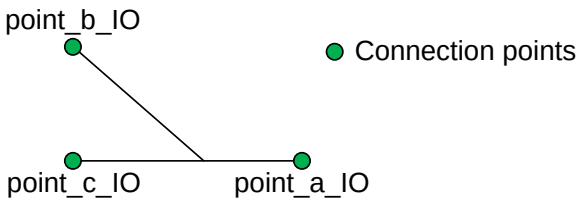


Figure 6. A symbolic representation of a railway point

The point operates in the following manner. When a train comes into one of the ports, the other ports cannot be occupied. Moreover, the motion from port point_b_IO to port point_c_IO and vise versa (see Fig. 6) is not allowed. For instance, if a train enters port point_b_IO, it can leave from port point_a_IO or from port point_b_IO (reverse motion).

A point does not have parameters but only behaves as described above. Thus, we will only show the machine that captures the aforementioned behavior and properties. The complete formal model of a railway point can be found in Appendix C.

We start the development of a specification of the point by introducing the ports and the modes. As mentioned above, there are three ports in a point. We assume that the values the ports can take range over the Boolean type (true, false), where true stands for a train present on the port and false, otherwise. Since the ports are bidirectional, the point has four modes: 0 – reading the state of the ports, 1 – a train comes from port a, 2 – a train comes from port b and 3 – a train comes from port c (Listing 15).

Listing 15. The variables of a railway point

```
machine RailwayPoint_Behaviour
variables
 railwayPoint_a_IO
 railwayPoint_b_IO
 railwayPoint_c_IO
 railwayPoint_mode
```

```
invariants
  railwayPoint_a_IO ∈ BOOL ∧ railwayPoint_b_IO ∈ BOOL ∧ railwayPoint_c_IO ∈ BOOL ∧ railwayPoint_mode ∈ 0..3
```

The properties of the point are captured as two invariants and two theorems (Listing 16). Particularly, if there is a train on port point_b_IO (see Fig. 6), port point_c_IO cannot have a train and vise versa (railwayPoint_b_IO = FALSE ∨ railwayPoint_c_IO = FALSE). However, if a train is long enough, it can occupy two ports (e.g., point_b_IO and point_a_IO) at the same time, which is postulated as **theorem** railwayPoint_a_IO = FALSE ∨ railwayPoint_b_IO = FALSE ∨ railwayPoint_c_IO = FALSE. Moreover, once the train has entered the point, it has to eventually leave it (railwayPoint_mode = 0 ⇒ railwayPoint_a_IO = railwayPoint_b_IO ∨ railwayPoint_a_IO = railwayPoint_c_IO). Finally, the point should always function (i.e., never terminates). This property is captured by the corresponding deadlock freedom theorem (**theorem** ((railwayPoint_mode = 1 ∧ railwayPoint_c_IO = FALSE) ∨ (railwayPoint_mode = 1 ∧ railwayPoint_b_IO = FALSE) ∨ ...).

Listing 16. The properties of a railway point

```
(railwayPoint_b_IO = FALSE ∨ railwayPoint_c_IO = FALSE) ∧
(railwayPoint_mode = 0 ⇒ railwayPoint_a_IO = railwayPoint_b_IO ∨ railwayPoint_a_IO = railwayPoint_c_IO) ∧
theorem (railwayPoint_a_IO = FALSE ∨ railwayPoint_b_IO = FALSE ∨ railwayPoint_c_IO = FALSE) ∧
theorem ((railwayPoint_mode = 1 ∧ railwayPoint_c_IO = FALSE) ∨
         (railwayPoint_mode = 1 ∧ railwayPoint_b_IO = FALSE) ∨
          (railwayPoint_mode = 2) ∨ (railwayPoint_mode = 3) ∨
          (∃a,b,c,m . a∈BOOL ∧ b∈BOOL ∧ c∈BOOL ∧ m∈0..3 ∧ (b=FALSE ∨ c=FALSE) ∧
           ((a=FALSE ∧ a=b ∧ b=c) ∨ (a=TRUE ∧ a=b) ∨ (a=TRUE ∧ a=c) ⇒ m = 0) ∧
            (¬ a=railwayPoint_a_IO ⇒ m=1) ∧ (¬ b=railwayPoint_b_IO ⇒ m=2) ∧ (¬ c=railwayPoint_c_IO ⇒ m=3))
```

The most complex event of the point model is the environment (Listing 17) due to the various limitations on the motion of trains. First, trains cannot occupy ports point_b_IO and point_c_IO at the same time (*railwayPoint_b_new* = FALSE ∨ *railwayPoint_c_new* = FALSE). Second, the direction from which a train comes into the point determines the mode. For instance, if there is no train on the point (*railwayPoint_a_new*=FALSE ∧ *railwayPoint_a_new*=*railwayPoint_b_new* ∧ *railwayPoint_b_new*=*railwayPoint_c_new*) or the train has cars attached to the train, so that it is long and moves from port point_a_IO to port point_b_IO (*railwayPoint_a_new* = TRUE ∧ *railwayPoint_a_new* = *railwayPoint_b_new*) or from port point_a_IO to port point_c_IO (*railwayPoint_a_new* = TRUE ∧ *railwayPoint_a_new* = *railwayPoint_c_new*), the railway point waits until the situation changes, i.e., the mode stays at reading the state of the ports (⇒ *railwayPoint_mode_new* = 0). On the other hand, if the train comes from port point_a_IO, for example, the mode changes accordingly (¬ *railwayPoint_a_new* = railwayPoint_a_IO ⇒ *railwayPoint_mode_new* = 1).

Listing 17. The update of the ports

```
event RailwayPoint_environment
  any
    railwayPoint_mode_new
    railwayPoint_a_new
    railwayPoint_b_new
    railwayPoint_c_new
  where
    railwayPoint_mode = 0 ∧ railwayPoint_mode_new ∈ 0..3 ∧ railwayPoint_a_new ∈ BOOL ∧
    railwayPoint_b_new ∈ BOOL ∧ railwayPoint_c_new ∈ BOOL ∧
    (railwayPoint_b_new = FALSE ∨ railwayPoint_c_new = FALSE) ∧
    (¬ railwayPoint_a_new = railwayPoint_a_IO ⇒ railwayPoint_mode_new = 1) ∧
    (¬ railwayPoint_b_new = railwayPoint_b_IO ⇒ railwayPoint_mode_new = 2) ∧
    (¬ railwayPoint_c_new = railwayPoint_c_IO ⇒ railwayPoint_mode_new = 3) ∧
   ((railwayPoint_a_new=FALSE ∧ railwayPoint_a_new=railwayPoint_b_new ∧ railwayPoint_b_new=railwayPoint_c_new) ∨
    (railwayPoint_a_new = TRUE ∧ railwayPoint_a_new = railwayPoint_b_new) ∨
    (railwayPoint_a_new = TRUE ∧ railwayPoint_a_new = railwayPoint_c_new) ⇒ railwayPoint_mode_new = 0)
```

```
  then
    railwayPoint_mode ≔ railwayPoint_mode_new || railwayPoint_a_IO ≔ railwayPoint_a_new ||
    railwayPoint_b_IO ≔ railwayPoint_b_new || railwayPoint_c_IO ≔ railwayPoint_c_new
end
```

If a train enters the point from the port point_a_IO, it can leave it from the port point_b_IO or point_c_IO (Listing 18 and Listing 19, respectively). The motion on the point is simply a change on the port. For instance, the motion from port point_a_IO to port point_b_IO is captured by the assignment railwayPoint_b_IO ≔ railwayPoint_a_IO in Listing 18 and from port point_a_IO to port point_c_IO by the assignment railwayPoint_c_IO ≔ railwayPoint_a_IO in Listing 19.

Listing 18. Train motion from a to b

```
event RailwayPoint_from_a_to_b
  where
    railwayPoint_mode = 1 ∧ railwayPoint_c_IO = FALSE
  then
    railwayPoint_mode ≔ 0 || railwayPoint_b_IO ≔ railwayPoint_a_IO
end
```

Listing 19. Train motion from a to c

```
event RailwayPoint_from_a_to_c
  where
    railwayPoint_mode = 1 ∧ railwayPoint_b_IO = FALSE
  then
    railwayPoint_mode ≔ 0 || railwayPoint_c_IO ≔ railwayPoint_a_IO
end
```

Another example of the train motion from port point_b_IO to port point_a_IO is shown in Listing 20. In this case, the mode of the point equals to 2.

Listing 20. Train motion from b to a

```
event RailwayPoint_from_b_to_a
  where
    railwayPoint_mode = 2
  then
    railwayPoint_mode ≔ 0 || railwayPoint_a_IO ≔ railwayPoint_b_IO
end
```

4.5. Railway components: a railway crossing

A railway crossing is another example of a component from the railway domain. The crossing has four bidirectional ports as shown in Fig. 7.

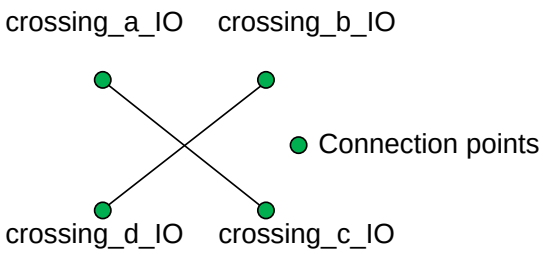


Figure 7. A visual represenation of a railway crossing

The crossing operates in the following manner. When a train enters port crossing_a_IO, for example, it can leave the crossing from the same port (inverse motion) or from port crossing_c_IO. In other words, the

motion from port crossing_a_IO to ports crossing_b_IO and crossing_d_IO is not allowed. The same property holds for other ports.

The specification of the crossing does not have any parameters similarly to the specification of a railway point. Thus, we only focus on the Event-B machine that captures the behaviour and whose complete code can be found in Appendix D.

We start the development of the crossing specification by introducing the ports and the modes (Listing 21). As mentioned above, the crossing has four ports (see Fig. 7). Since the ports of the crossing are bidirectional, there are five modes: 0 – reading the state of the ports, 1 – a train goes from port a to port c, 2 – a train moves from port b to port d, 3 – a train crosses the railway from port c to port a and 4 – a train goes from port d to port b.

Listing 21. Variables of the crossing specification

```
machine RailwayCrossing_Behaviour
variables
  railwayCrossing_a_IO
  railwayCrossing_b_IO
  railwayCrossing_c_IO
  railwayCrossing_d_IO
  railwayCrossing_mode
invariants
  railwayCrossing_a_IO ∈ BOOL ∧ railwayCrossing_b_IO ∈ BOOL ∧ railwayCrossing_c_IO ∈ BOOL ∧
  railwayCrossing_d_IO ∈ BOOL ∧ railwayCrossing_mode ∈ 0..4
```

The properties of the crossing that have to be satisfied are presented in Listing 22. Firstly, there are limitation on the movements within the crossing as explained above. For instance, the ports a and b cannot be occupied simultaneously (railwayCrossing_a_IO = FALSE ∨ railwayCrossing_b_IO = FALSE). Whenever a train enters a crossing, it has to leave it eventually (railwayCrossing_mode = 0 ⇒ railwayCrossing_a_IO = railwayCrossing_c_IO ∨ railwayCrossing_b_IO = railwayCrossing_d_IO). Finally, the crossing has to be non-terminating as any other component, which is captured by the deadlock freedom theorem.

Listing 22. Properties of the crossing

```
(railwayCrossing_mode=0⇒railwayCrossing_a_IO=railwayCrossing_c_IO ∨ railwayCrossing_b_IO=railwayCrossing_d_IO) ∧
(railwayCrossing_a_IO = FALSE ∨ railwayCrossing_b_IO = FALSE) ∧
(railwayCrossing_b_IO = FALSE ∨ railwayCrossing_c_IO = FALSE) ∧
(railwayCrossing_c_IO = FALSE ∨ railwayCrossing_d_IO = FALSE) ∧
(railwayCrossing_d_IO = FALSE ∨ railwayCrossing_a_IO = FALSE) ∧
 theorem (railwayCrossing_mode = 1 ∧ railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE) ∨
         (railwayCrossing_mode = 3 ∧ railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE) ∨
         (railwayCrossing_mode = 2 ∧ railwayCrossing_a_IO = FALSE ∧ railwayCrossing_c_IO = FALSE) ∨
         (railwayCrossing_mode = 4 ∧ railwayCrossing_a_IO = FALSE ∧ railwayCrossing_c_IO = FALSE) ∨
         (∃mode, a, b, c, d . mode ∈ 0..4 ∧ a ∈ BOOL ∧ b ∈ BOOL ∧ c ∈ BOOL ∧ d ∈ BOOL ∧
          (a = FALSE ∨ b = FALSE) ∧ (b = FALSE ∨ c = FALSE) ∧ (c = FALSE ∨ d = FALSE) ∧ (d = FALSE ∨ a = FALSE) ∧
          (¬ a = railwayCrossing_a_IO ⇒ mode = 1) ∧ (¬ b = railwayCrossing_b_IO ⇒ mode = 2) ∧
          (¬ c = railwayCrossing_c_IO ⇒ mode = 3) ∧ (¬ d = railwayCrossing_d_IO ⇒ mode = 4) ∧
          ((a = FALSE ∧ a = b ∧ b = c ∧ c = d) ∨ (a = TRUE ∧ a = c) ∨ (b = TRUE ∧ b = d) ⇒ mode = 0) )
```

Initially, there are no trains on the crossing and the mode is set to the reading of the status of the ports. Similarly to the railway point, the most complex event of the crossing specification is environment (Listing 23). First, it needs to consider the fact that only one train can be on the crossing at a time (e.g., *railwayCrossing_a_new* = FALSE ∨ *railwayCrossing_b_new* = FALSE). Second, the motion direction of the train affects the mode (e.g., ¬ *railwayCrossing_a_new* = railwayCrossing_a_IO ⇒ *railwayCrossing_mode_new* = 1). Finally, if there is no train on the crossing or the train long enough to occupy two ports, the mode stays at 0, i.e., reading the

status of the ports (*railwayCrossing_a_new* = FALSE ∧ *railwayCrossing_a_new* = *railwayCrossing_b_new* ∧ *railwayCrossing_b_new* = *railwayCrossing_c_new* ... ⇒ *railwayCrossing_mode_new* = 0).

Listing 23. Environment event of the crossing specification

```
event RailwayCrossing_environment
 any
   railwayCrossing_mode_new
   railwayCrossing_a_new
   railwayCrossing_b_new
   railwayCrossing_c_new
   railwayCrossing_d_new
 where
   railwayCrossing_mode = 0 ∧ railwayCrossing_mode_new ∈ 0..4 ∧ railwayCrossing_a_new ∈ BOOL ∧
   railwayCrossing_b_new ∈ BOOL ∧ railwayCrossing_c_new ∈ BOOL ∧ railwayCrossing_d_new ∈ BOOL ∧
   (railwayCrossing_a_new = FALSE ∨ railwayCrossing_b_new = FALSE) ∧
   (railwayCrossing_b_new = FALSE ∨ railwayCrossing_c_new = FALSE) ∧
   (railwayCrossing_c_new = FALSE ∨ railwayCrossing_d_new = FALSE) ∧
   (railwayCrossing_d_new = FALSE ∨ railwayCrossing_a_new = FALSE) ∧
   (¬ railwayCrossing_a_new = railwayCrossing_a_IO ⇒ railwayCrossing_mode_new = 1) ∧
   (¬ railwayCrossing_b_new = railwayCrossing_b_IO ⇒ railwayCrossing_mode_new = 2) ∧
   (¬ railwayCrossing_c_new = railwayCrossing_c_IO ⇒ railwayCrossing_mode_new = 3) ∧
   (¬ railwayCrossing_d_new = railwayCrossing_d_IO ⇒ railwayCrossing_mode_new = 4) ∧
 ((railwayCrossing_a_new = FALSE ∧ railwayCrossing_a_new = railwayCrossing_b_new ∧
 railwayCrossing_b_new = railwayCrossing_c_new ∧ railwayCrossing_c_new = railwayCrossing_d_new) ∨
 (railwayCrossing_a_new = TRUE ∧ railwayCrossing_a_new = railwayCrossing_c_new) ∨
 (railwayCrossing_b_new = TRUE ∧ railwayCrossing_b_new = railwayCrossing_d_new) ⇒ railwayCrossing_mode_new = 0)
 then
   railwayCrossing_mode ≔ railwayCrossing_mode_new || railwayCrossing_a_IO ≔ railwayCrossing_a_new ||
   railwayCrossing_b_IO ≔ railwayCrossing_b_new || railwayCrossing_c_IO ≔ railwayCrossing_c_new ||
   railwayCrossing_d_IO ≔ railwayCrossing_d_new
end
```

Whenever a train moves from one port to another one, the corresponding properties have to hold. Listing 24 and Listing 25 show examples of the train motion from port a to port c and from port c to port a, respectively. Notice that when the train moves from port a to port c, for example (Listing 24), the ports b and d have to be free of trains (railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE).

Listing 24. The train motion from port a to port c

```
event RailwayCrossing_from_a_to_c
 where
   railwayCrossing_mode = 1 ∧ railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE
 then
   railwayCrossing_mode ≔ 0 || railwayCrossing_c_IO ≔ railwayCrossing_a_IO
end
```

Listing 25. The train motion from port c to port a

```
event RailwayCrossing_from_c_to_a
 where
   railwayCrossing_mode = 3 ∧ railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE
 then
   railwayCrossing_mode ≔ 0 || railwayCrossing_a_IO ≔ railwayCrossing_c_IO
end
```

4.6. Formal Model of the Generic Component

Often, the development of a system requires decisions on which components have to be used in the system. However, this information may not be known a priori, in which case the designers typically use a placeholder (i.e., a box with input and output) instead of a specific component. In case of a model, this placeholder can be captured by the specification of a generic component.

A generic component is a component whose precise functionality is unspecified, but there is a relation between its input and output (Fig. 8). Since any type (e.g., Boolean, numbers) can be modelled or encoded with numbers, we assume that the input and the output of the generic component are subsets of integers. The complete model of the generic component can be found in 0.

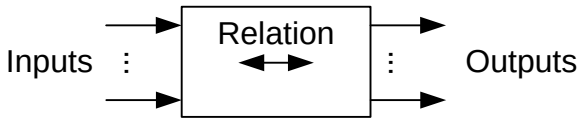


Figure 8. A graphical representation of the generic component

The model of the generic component is as follows. The component has an input and output as well as the mode and a relation as illustrated in Listing 26. The theorem defined in the machine eases the proving effort related to the properties of power sets.

Listing 26. The definitions of the generic component

```
machine GenericComponent_Behaviour
variables
  GenericComponent_I
  GenericComponent_O
  GenericComponent_IOrelation
  GenericComponent_mode
invariants
  theorem (∀ps,s·ps ∈ ℙ1(ℤ) ∧ finite(ps) ∧ s ∈ ℙ(ps) ∧ card(s) = card(ps) ⇒ s = ps) ∧
  GenericComponent_I ∈ ℙ1(ℤ) ∧ GenericComponent_O ∈ ℙ1(ℤ) ∧ GenericComponent_mode ∈ 0..1 ∧
  GenericComponent_IOrelation ∈ GenericComponent_I ↔ GenericComponent_O
```

The generic component has the following properties (Listing 27). First, the component has a finite set of values it can take as the input and produce as the output (finite(GenericComponent_I) ∧ finite(GenericComponent_O)). Second, although the relation between the input and the output is generic, it has to be defined on the input values, such that the component returns a non-empty set as the result. In other words, the relation is total (dom(GenericComponent_IOrelation) = GenericComponent_I) and surjective (ran(GenericComponent_IOrelation) = GenericComponent_O). Finally, the component produces the output according the input and the relation between the input and the output (GenericComponent_mode = 0 ⇒ GenericComponent_O = GenericComponent_IOrelation[GenericComponent_I]).

Listing 27. The vital properties of the generic component

```
finite(GenericComponent_I) ∧ dom(GenericComponent_IOrelation) = GenericComponent_I ∧
finite(GenericComponent_O) ∧ ran(GenericComponent_IOrelation) = GenericComponent_O ∧
(GenericComponent_mode = 0 ⇒ GenericComponent_O = GenericComponent_IOrelation[GenericComponent_I])
```

Initially, the component is set to the reading mode and the input, output and relation are non-deterministically assigned some values that preserve the invariants (Listing 28).

Listing 28. Initialization of the state variables

```
event INITIALISATION
  then
```

```
    GenericComponent_mode ≔ 0 ||
    GenericComponent_I, GenericComponent_O, GenericComponent_IOrelation :|
                    GenericComponent_I' ∈ {i | i ∈ ℙ1(ℤ) ∧ finite(i)} ∧
                    GenericComponent_O' ∈ {o | o ∈ ℙ1(ℤ) ∧ finite(o)} ∧
                    GenericComponent_IOrelation' ∈ GenericComponent_I' ↔ GenericComponent_O' ∧
                    dom(GenericComponent_IOrelation') = GenericComponent_I' ∧
                    ran(GenericComponent_IOrelation') = GenericComponent_O'
end
```

As mentioned above, the component has two operations: reading the input and producing the output. These are simple operations shown in Listing 29 and Listing 30, respectively. Notice that although these operations have the deterministic behaviour, this model allows its refinement in various ways depending on the design decisions made during the system development. Notice also that one can develop a generic component whose ports are bidirectional considering the described models. Since this is a rather simple task, we omit the specification of this component, but it is present in the library.

Listing 29. Reading the input

```
event GenericComponent_environment // This event is needed to show that a component reads the inputs
  where
    GenericComponent_mode = 0
  then
    GenericComponent_mode ≔ 1 || GenericComponent_I ≔ dom(GenericComponent_IOrelation)
end
```

Listing 30. Producing the output

```
event GenericComponent_process_inputs // This event illustrates the production of the outputs
  where
    GenericComponent_mode = 1
  then
    GenericComponent_mode ≔ 0 || GenericComponent_O ≔ GenericComponent_IOrelation[GenericComponent_I]
end
```

5. Conclusion and future work

We presented a subset of formally developed parameterized components, each of which has a unique visual symbol. The correctness of the formal components is supported by the proofs obtained by the use of the Event-B mathematical engine. Every component can be instantiated and reused in various applications depending on the requirements. Visual design structures specifications and facilitates scalability of the rigorous development. In addition, it enhances the communication between the developer and a customer.

The library of the components indeed speeds up and eases the development process due to the possibility of picking and placing (instantiating) a component whenever needed. On the other hand, the components need to be properly/correctly connected with each other in order to form a system and to carry out the mission according to the requirements. Currently, we are working on the mechanisms of connecting the components together considering the stepwise refinement approach.

The library of the formal components presented in this paper is certainly not complete. The more components are present in the library, the more diverse applications can be visually and rigorously developed. Thus, one direction of our work is to extend the formal library with various components from different application domains.

Acknowledgment

The authors would like to thank Dr. Marta Olszewska and Dr. Andrew Edmunds for the fruitful discussions on the topic of this paper.

References

[1] J.-R. Abrial, Modeling in Event-B: System and Software Engineering. Cambridge: Cambridge University Press, 2010.

[2] R. J. Back and J. Wright, Refinement Calculus: A Systematic Introduction, Springer-Verlag, 1998.

[3] L. Ladenberger, J. Bendisposto, M. Leuschel, Visualising Event-B Models with B-Motion Studio, Proceedings of Formal Methods for Industrial Critical Systems (FMICS), LNCS: Springer Berlin Heidelberg, pp. 202-204, 2009.

[4] BMotion Studio for ProB Handbook, 2015. Available: http://nightly.cobra.cs.uni-duesseldorf.de/bmotion/bmotion-prob-handbook/nightly/html/index.html. Visualising Event-B Models with B-Motion Studio

[5] M. Leuschel, M. Butler, ProB: A Model Checker for B, Proc. FME, Springer, vol. 2805, 2003, p. 855-874.

[6] C. Snook, M. Butler, UML-B: Formal Modeling and Design Aided by UML, ACM Transactions on Software Engineering and Methodology, Vol. 15(1), pp. 92–122, 2006.

[7] G. Booch, I. Jacobson, J. Rumbaugh, The unified modeling language – a reference manual, 2nd edition, Addison-Wesley, p. 721, 2004.

[8] S. Schneider, The B-method: An Introduction, Basingstoke: Palgrave, p. 370, 2001.

[9] C. Snook, M. Butler, UML-B and Event-B: an integration of languages and tools, Proceedings of IASTED International Conference on Software Engineering, pp. 12, 2008.

[10] S. Ostroumov, L. Tsiopoulos, J. Plosila, K. Sere, Generation of Structural VHDL Code with Library Components From Formal Event-B Models, In Proceedings of Euromicro Conference on Digital System Design, IEEE Conference Publishing Services (CPS), pp. 111-118, 2013.

[11] IEEE Standard VHDL Language Reference Manual, IEEE 1076, 2008.

[12] M. Butler, E. Sekerinski, K. Sere, An Action System Approach to the Steam Boiler Problem, Formal Methods For Industrial Applications, Vol. 1165, LNCS: Springer-Verlag, pp. 129-148, 1996.

[13] RODIN, 2014. Available: http://sourceforge.net/projects/rodin-b-sharp/.

[14] C. Métayer, J.-R. Abrial, L. Voisin, Deliverables, Rigorous Open Development Environment for Complex Systems, 2005. Available: http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf.

[15] K. Robinson, System Modelling & Designing using Event-B, 2010. Available: http://wiki.event-b.org/images/SM%26D-KAR.pdf.

Appendix A

The complete model of a valve

```
context Valve_parameters
constants valve_diameter_min_val valve_diameter_max_val valve_CONTROL valve_rate
axioms
  @valve_axm_0 valve_diameter_min_val = 0 // If position of a valve is at minimum, the valve is fully closed (0% open)
  @valve_axm_1 valve_diameter_max_val ∈ ℕ1 // On contrary, maximum means that the valve is fully open (100% open)
  @valve_axm_2 valve_CONTROL = {−1,0,1} // -1 - closing, 0 - OFF, 1 - opening
  @valve_axm_3 valve_rate ∈ ℕ1 // The rate showing how fast the valve opens
  @valve_axm_4 valve_rate ≤ valve_diameter_max_val − valve_diameter_min_val // The rate must not exceed this value
end



machine Valve_Behaviour sees Valve_parameters
variables valve_control_I valve_flow_I valve_flow_O valve_mode valve_position
invariants
  @valve_inv_0 valve_control_I ∈ valve_CONTROL // Control input for the valve: -1 - close, 0 - OFF, 1 - open
  @valve_inv_1 valve_flow_I ∈ valve_diameter_min_val..valve_diameter_max_val // The flow of fluid coming into the valve
  @valve_inv_2 valve_flow_O ∈ valve_diameter_min_val..valve_diameter_max_val // The flow of fluid coming out the valve
  // To obtain the deterministic behaviour of the component, we use an internal variable that specifies the mode
  @valve_inv_3 valve_mode ∈ 0..1
  // The current state of the valve
  @valve_inv_4 valve_position ∈ valve_diameter_min_val..valve_diameter_max_val
  // The output flow cannot be stronger than the input flow
  @valve_inv_10 valve_mode = 0 ⇒ valve_flow_O ≤ valve_flow_I
  // The output flow cannot be larger than the opening of the valve
  @valve_inv_11 valve_flow_O ≤ valve_position
  // The non-termination property
  theorem @valve_DLF valve_mode = 0 ∨
                (valve_control_I = 1 ∧ valve_position + valve_rate ≤ valve_diameter_max_val ∧ valve_mode = 1) ∨
                (valve_control_I = −1 ∧ valve_position − valve_rate ≥ valve_diameter_min_val ∧ valve_mode = 1) ∨
                ((valve_control_I = 0 ∨
                (valve_position − valve_rate < valve_diameter_min_val ∧ valve_control_I = −1) ∨
                (valve_position + valve_rate > valve_diameter_max_val ∧ valve_control_I = 1)) ∧ valve_mode = 1)
events
  event INITIALISATION // Initially, the valve is closed
    then
      @valve_act_0 valve_control_I ≔ 0
      @valve_act_1 valve_flow_I :∈ valve_diameter_min_val..valve_diameter_max_val
      @valve_act_2 valve_flow_O ≔ valve_diameter_min_val
      @valve_act_3 valve_mode ≔ 0
      @valve_act_4 valve_position ≔ valve_diameter_min_val
  end

  event valve_environment
    where
      @grd_0 valve_mode = 0 // One can also have FALSE instead of 0
    then
      @act_0 valve_mode ≔ 1 // and TRUE instead of 1, if there are two alternating states
      @act_1 valve_control_I :∈ valve_CONTROL
      @act_2 valve_flow_I :∈ valve_diameter_min_val..valve_diameter_max_val
  end
```

```
// While the command is open, the valve should be opening with some rate
event valve_opening
  any valve_flow_O_new
  where
    @grd_0 valve_control_I = 1 // If the command is to open the valve
    @grd_1 valve_mode = 1
    @grd_2 valve_position + valve_rate ≤ valve_diameter_max_val // and it is not completely open
    // The valve opens and allows the flow to go through with some rate
    @grd_3 valve_position + valve_rate < valve_flow_I ⇒ valve_flow_O_new = valve_position + valve_rate
    // but the output flow cannot be stronger than the input one, even if the valve is completely open
    @grd_4 valve_position + valve_rate ≥ valve_flow_I ⇒ valve_flow_O_new = valve_flow_I
  then
    @act_0 valve_flow_O ≔ valve_flow_O_new
    @act_1 valve_mode ≔ 0
    @act_2 valve_position ≔ valve_position + valve_rate
end

// While the command is close, the valve should be closing with some rate
event valve_closing
  any valve_flow_O_new
  where
    @grd_0 valve_control_I = −1 // If the command is to close the valve
    @grd_1 valve_position − valve_rate ≥ valve_diameter_min_val // and the valve is not completely closed yet
    @grd_2 valve_mode = 1
    // The valve closes and decreases the flow with some rate
    @grd_3 valve_position − valve_rate ≤ valve_flow_I ⇒ valve_flow_O_new = valve_position − valve_rate
    // but if it is open more than the input flow is, the output flow should be updated accordingly
    @grd_4 valve_position − valve_rate > valve_flow_I ⇒ valve_flow_O_new = valve_flow_I
  then
    @act_0 valve_flow_O ≔ valve_flow_O_new
    @act_1 valve_mode ≔ 0
    @act_2 valve_position ≔ valve_position − valve_rate
end

// The valve is stopped in all other cases not captured by the aforementioned events
event valve_stop
  any valve_flow_O_new
  where
    @grd_0 valve_control_I = 0 ∨
          (valve_position − valve_rate < valve_diameter_min_val ∧ valve_control_I = −1) ∨
          (valve_position + valve_rate > valve_diameter_max_val ∧ valve_control_I = 1)
    @grd_1 valve_mode = 1
    @grd_2 valve_flow_I < valve_flow_O ⇒ valve_flow_O_new = valve_flow_I
    @grd_3 valve_flow_I ≥ valve_flow_O ⇒ valve_flow_O_new = valve_flow_O
  then
    @act_0 valve_mode ≔ 0
    @act_1 valve_flow_O ≔ valve_flow_O_new
end
end
```

Appendix B

The complete model of the cylinder

```
context Cylinder_parameters
constants
  cylinder_input_diameter_min_val
  cylinder_input_diameter_max_val
  cylinder_cap_pos
  cylinder_head_pos
axioms
  // 0 stands for no liquid flowing into the cylinder (0% open)
  @cylinder_axm_0 cylinder_input_diameter_min_val = 0
  // 100 stands for maximum velocity the piston can move inside the cylinder (100% open)
  @cylinder_axm_1 cylinder_input_diameter_max_val ∈ ℕ1
  // We assume that when the piston is in the leftmost position (at the cap), it is at 0 (%) – completely retracted
  @cylinder_axm_2 cylinder_cap_pos = 0
  // On contrary, when the piston is in the rightmost position (at the head), it is at 100 (%) – completely extended
  @cylinder_axm_3 cylinder_head_pos ∈ ℕ1
end
```

```
machine Cylinder_behaviour sees Cylinder_parameters
variables
  cylinder_flow_cap_I
  cylinder_flow_head_I
  cylinder_piston_position_O
  cylinder_mode
invariants
  // Current position of the piston in the cylinder
  @cylinder_inv_0 cylinder_piston_position_O ∈ cylinder_cap_pos..cylinder_head_pos
  // Input to move the piston to the right
  @cylinder_inv_1 cylinder_flow_cap_I ∈ cylinder_input_diameter_min_val..cylinder_input_diameter_max_val
  // Input to move the piston to the left
  @cylinder_inv_2 cylinder_flow_head_I ∈ cylinder_input_diameter_min_val..cylinder_input_diameter_max_val
  @cylinder_inv_3 cylinder_mode ∈ 0..1
  // Deadlock freedom – non-termination
  theorem @cylinder_DLF cylinder_mode = 0 ∨
                  (cylinder_flow_cap_I > cylinder_input_diameter_min_val ∧
                   cylinder_flow_cap_I > cylinder_flow_head_I ∧
                   cylinder_piston_position_O + cylinder_flow_cap_I − cylinder_flow_head_I ≤ cylinder_head_pos ∧
                   cylinder_mode = 1) ∨
                  (cylinder_flow_head_I > cylinder_input_diameter_min_val ∧
                   cylinder_flow_head_I > cylinder_flow_cap_I ∧
                   cylinder_cap_pos ≤ cylinder_piston_position_O + cylinder_flow_cap_I − cylinder_flow_head_I ∧
                   cylinder_mode = 1) ∨
                  (∃cylinder_rate . cylinder_rate = cylinder_flow_cap_I − cylinder_flow_head_I ∧ cylinder_mode = 1 ∧
                   (cylinder_flow_head_I = cylinder_flow_cap_I ∨
                    cylinder_piston_position_O + cylinder_rate < cylinder_cap_pos ∨
                    cylinder_piston_position_O + cylinder_rate > cylinder_head_pos))
events
  event INITIALISATION // At the beginning, the piston does not move and is at some position
    then
      @cylinder_act_0 cylinder_piston_position_O :∈ cylinder_cap_pos..cylinder_head_pos
```

```
    @cylinder_act_1 cylinder_flow_cap_I ≔ cylinder_input_diameter_min_val
    @cylinder_act_2 cylinder_flow_head_I ≔ cylinder_input_diameter_min_val
    @cylinder_act_3 cylinder_mode ≔ 0
  end

  event cylinder_environment
   where
    @grd_0 cylinder_mode = 0
   then
    @act_0 cylinder_flow_cap_I :∈ cylinder_input_diameter_min_val..cylinder_input_diameter_max_val
    @act_1 cylinder_flow_head_I :∈ cylinder_input_diameter_min_val..cylinder_input_diameter_max_val
    @act_2 cylinder_mode ≔1
  end

  event cylinder_extending // Extending stands for the motion of the piston to the right
   any cylinder_rate
   where
    @grd_0 cylinder_rate = cylinder_flow_cap_I − cylinder_flow_head_I
    // If there is a flow of fluid to move the piston to the right
    @grd_1 cylinder_flow_cap_I > cylinder_input_diameter_min_val
    @grd_2 cylinder_flow_cap_I > cylinder_flow_head_I // Moreover, if this flow is stronger than the one to the left
    @grd_3 cylinder_piston_position_O + cylinder_rate ≤ cylinder_head_pos // and there is a space for piston to move
    @grd_4 cylinder_mode = 1
   then
    // The piston moves to the right with the velocity equal to the difference of flows
    @act_0 cylinder_piston_position_O ≔ cylinder_piston_position_O + cylinder_rate
    @act_1 cylinder_mode ≔ 0
  end

  event cylinder_retracting // Retracting stands for the motion of the piston to the left
   any cylinder_rate
   where
    @grd_0 cylinder_rate = cylinder_flow_cap_I − cylinder_flow_head_I
    // Similarly, if there is a flow of fluid to move piston to the left
    @grd_1 cylinder_flow_head_I > cylinder_input_diameter_min_val
    @grd_2 cylinder_flow_head_I > cylinder_flow_cap_I // and it is stronger than the one to move piston to the right
    @grd_3 cylinder_cap_pos ≤ cylinder_piston_position_O + cylinder_rate // There is a space for a piston to move
    @grd_4 cylinder_mode = 1
   then
    @act_0 cylinder_mode ≔ 0
    // The piston moves to the left with the velocity equal to the difference of flows
    @act_1 cylinder_piston_position_O ≔ cylinder_piston_position_O + cylinder_rate
  end

  // When the incoming flows are the same or there is no place for the piston to move, it is simply stopped
  event cylinder_stop
   any cylinder_rate
   where
    @grd_0 cylinder_rate = cylinder_flow_cap_I − cylinder_flow_head_I
    @grd_1 cylinder_mode = 1
    @grd_2 cylinder_flow_head_I = cylinder_flow_cap_I ∨
              cylinder_piston_position_O + cylinder_rate < cylinder_cap_pos ∨
              cylinder_piston_position_O + cylinder_rate > cylinder_head_pos
   then
    @act_0 cylinder_mode ≔ 0
  end
 end
```

Appendix C

The complete model of the railway point

```
machine RailwayPoint_Behaviour
variables
  railwayPoint_a_IO
  railwayPoint_b_IO
  railwayPoint_c_IO
  railwayPoint_mode
invariants
  @railwayPoint_inv_0 railwayPoint_a_IO ∈ BOOL
  @railwayPoint_inv_1 railwayPoint_b_IO ∈ BOOL
  @railwayPoint_inv_2 railwayPoint_c_IO ∈ BOOL
  @railwayPoint_inv_3 railwayPoint_mode ∈ 0..3
  @railwayPoint_inv_10 railwayPoint_mode = 0 ⇒
                                        railwayPoint_a_IO = railwayPoint_b_IO ∨ railwayPoint_a_IO = railwayPoint_c_IO
  @railwayPoint_inv_11 railwayPoint_b_IO = FALSE ∨ railwayPoint_c_IO = FALSE
  theorem @railwayPoint_inv_12 railwayPoint_a_IO = FALSE ∨ railwayPoint_b_IO = FALSE ∨ railwayPoint_c_IO = FALSE
  theorem @railwayPoint_DLF
          (railwayPoint_mode = 1 ∧ railwayPoint_c_IO = FALSE) ∨
          (railwayPoint_mode = 1 ∧ railwayPoint_b_IO = FALSE) ∨
          (railwayPoint_mode = 2) ∨
          (railwayPoint_mode = 3) ∨
          (∃a,b,c,m . a∈BOOL ∧ b∈BOOL ∧ c∈BOOL ∧ m∈0..3 ∧
                  (b=FALSE ∨ c=FALSE) ∧
                  ((a=FALSE ∧ a=b ∧ b=c) ∨ (a=TRUE ∧ a=b) ∨ (a=TRUE ∧ a=c) ⇒ m = 0) ∧
                  (¬ a=railwayPoint_a_IO ⇒ m=1) ∧
                  (¬ b=railwayPoint_b_IO ⇒ m=2) ∧
                  (¬ c=railwayPoint_c_IO ⇒ m=3))
events
  event INITIALISATION
    then
      @railwayPoint_act_0 railwayPoint_a_IO ≔ FALSE
      @railwayPoint_act_1 railwayPoint_b_IO ≔ FALSE
      @railwayPoint_act_2 railwayPoint_c_IO ≔ FALSE
      @railwayPoint_act_3 railwayPoint_mode ≔ 0
  end

  event RailwayPoint_environment
    any
      railwayPoint_mode_new
      railwayPoint_a_new
      railwayPoint_b_new
      railwayPoint_c_new
    where
      @grd_0 railwayPoint_mode = 0
      @grd_1 railwayPoint_mode_new ∈ 0..3
      @grd_2 railwayPoint_a_new ∈ BOOL
      @grd_3 railwayPoint_b_new ∈ BOOL
      @grd_4 railwayPoint_c_new ∈ BOOL
      @grd_5 railwayPoint_b_new = FALSE ∨ railwayPoint_c_new = FALSE
      @grd_6 (railwayPoint_a_new = FALSE ∧
              railwayPoint_a_new = railwayPoint_b_new ∧ railwayPoint_b_new = railwayPoint_c_new) ∨
             (railwayPoint_a_new = TRUE ∧ railwayPoint_a_new = railwayPoint_b_new) ∨
             (railwayPoint_a_new = TRUE ∧ railwayPoint_a_new = railwayPoint_c_new) ⇒ railwayPoint_mode_new = 0
      @grd_7 ¬ railwayPoint_a_new = railwayPoint_a_IO ⇒ railwayPoint_mode_new = 1
```

```
    @grd_8 ¬ railwayPoint_b_new = railwayPoint_b_IO ⇒ railwayPoint_mode_new = 2
    @grd_9 ¬ railwayPoint_c_new = railwayPoint_c_IO ⇒ railwayPoint_mode_new = 3
  then
    @act_0 railwayPoint_mode ≔ railwayPoint_mode_new
    @act_1 railwayPoint_a_IO ≔ railwayPoint_a_new
    @act_2 railwayPoint_b_IO ≔ railwayPoint_b_new
    @act_3 railwayPoint_c_IO ≔ railwayPoint_c_new
 end

 event RailwayPoint_from_a_to_b
  where
    @grd_0 railwayPoint_mode = 1
    @grd_1 railwayPoint_c_IO = FALSE
  then
    @act_0 railwayPoint_mode ≔ 0
    @act_1 railwayPoint_b_IO ≔ railwayPoint_a_IO
 end

 event RailwayPoint_from_a_to_c
  where
    @grd_0 railwayPoint_mode = 1
    @grd_1 railwayPoint_b_IO = FALSE
  then
    @act_0 railwayPoint_mode ≔ 0
    @act_1 railwayPoint_c_IO ≔ railwayPoint_a_IO
 end

 event RailwayPoint_from_b_to_a
  where
    @grd_0 railwayPoint_mode = 2
  then
    @act_0 railwayPoint_mode ≔ 0
    @act_1 railwayPoint_a_IO ≔ railwayPoint_b_IO
 end

 event RailwayPoint_from_c_to_a
  where
    @grd_0 railwayPoint_mode = 3
  then
    @act_0 railwayPoint_mode ≔ 0
    @act_1 railwayPoint_a_IO ≔ railwayPoint_c_IO
 end
end
```

Appendix D

The complete model of the railway crossing

```
machine RailwayCrossing_Behaviour
variables
  railwayCrossing_a_IO
  railwayCrossing_b_IO
  railwayCrossing_c_IO
  railwayCrossing_d_IO
  railwayCrossing_mode
invariants
  @railwayCrossing_inv_0 railwayCrossing_a_IO ∈ BOOL
  @railwayCrossing_inv_1 railwayCrossing_b_IO ∈ BOOL
  @railwayCrossing_inv_2 railwayCrossing_c_IO ∈ BOOL
  @railwayCrossing_inv_3 railwayCrossing_d_IO ∈ BOOL
  @railwayCrossing_inv_4 railwayCrossing_mode ∈ 0..4
  @railwayCrossing_inv_10 railwayCrossing_mode = 0 ⇒
                                railwayCrossing_a_IO = railwayCrossing_c_IO ∨ railwayCrossing_b_IO = railwayCrossing_d_IO
  @railwayCrossing_inv_11 railwayCrossing_a_IO = FALSE ∨ railwayCrossing_b_IO = FALSE
  @railwayCrossing_inv_12 railwayCrossing_b_IO = FALSE ∨ railwayCrossing_c_IO = FALSE
  @railwayCrossing_inv_13 railwayCrossing_c_IO = FALSE ∨ railwayCrossing_d_IO = FALSE
  @railwayCrossing_inv_14 railwayCrossing_d_IO = FALSE ∨ railwayCrossing_a_IO = FALSE
  theorem @railwayCrossing_DLF
          (railwayCrossing_mode = 1 ∧ railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE) ∨
          (railwayCrossing_mode = 3 ∧ railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE) ∨
          (railwayCrossing_mode = 2 ∧ railwayCrossing_a_IO = FALSE ∧ railwayCrossing_c_IO = FALSE) ∨
          (railwayCrossing_mode = 4 ∧ railwayCrossing_a_IO = FALSE ∧ railwayCrossing_c_IO = FALSE) ∨
          (∃mode, a, b, c, d . mode ∈ 0..4 ∧ a ∈ BOOL ∧ b ∈ BOOL ∧ c ∈ BOOL ∧ d ∈ BOOL ∧
           (a = FALSE ∨ b = FALSE) ∧ (b = FALSE ∨ c = FALSE) ∧ (c = FALSE ∨ d = FALSE) ∧ (d = FALSE ∨ a = FALSE) ∧
           (¬ a = railwayCrossing_a_IO ⇒ mode = 1) ∧ (¬ b = railwayCrossing_b_IO ⇒ mode = 2) ∧
           (¬ c = railwayCrossing_c_IO ⇒ mode = 3) ∧ (¬ d = railwayCrossing_d_IO ⇒ mode = 4) ∧
           ((a = FALSE ∧ a = b ∧ b = c ∧ c = d) ∨ (a = TRUE ∧ a = c) ∨ (b = TRUE ∧ b = d) ⇒ mode = 0) )
events
  event INITIALISATION
    then
      @railwayCrossing_act_0 railwayCrossing_a_IO ≔ FALSE
      @railwayCrossing_act_1 railwayCrossing_b_IO ≔ FALSE
      @railwayCrossing_act_2 railwayCrossing_c_IO ≔ FALSE
      @railwayCrossing_act_3 railwayCrossing_d_IO ≔ FALSE
      @railwayCrossing_act_4 railwayCrossing_mode ≔ 0
  end

  event RailwayCrossing_environment
    any
      railwayCrossing_mode_new
      railwayCrossing_a_new
      railwayCrossing_b_new
      railwayCrossing_c_new
      railwayCrossing_d_new
    where
      @grd_0 railwayCrossing_mode = 0
      @grd_1 railwayCrossing_mode_new ∈ 0..4
      @grd_2 railwayCrossing_a_new ∈ BOOL
      @grd_3 railwayCrossing_b_new ∈ BOOL
      @grd_4 railwayCrossing_c_new ∈ BOOL
      @grd_5 railwayCrossing_d_new ∈ BOOL
```

```
    @grd_6 railwayCrossing_a_new = FALSE ∨ railwayCrossing_b_new = FALSE
    @grd_7 railwayCrossing_b_new = FALSE ∨ railwayCrossing_c_new = FALSE
    @grd_8 railwayCrossing_c_new = FALSE ∨ railwayCrossing_d_new = FALSE
    @grd_9 railwayCrossing_d_new = FALSE ∨ railwayCrossing_a_new = FALSE
    @grd_10 ¬ railwayCrossing_a_new = railwayCrossing_a_IO ⇒ railwayCrossing_mode_new = 1
    @grd_11 ¬ railwayCrossing_b_new = railwayCrossing_b_IO ⇒ railwayCrossing_mode_new = 2
    @grd_12 ¬ railwayCrossing_c_new = railwayCrossing_c_IO ⇒ railwayCrossing_mode_new = 3
    @grd_13 ¬ railwayCrossing_d_new = railwayCrossing_d_IO ⇒ railwayCrossing_mode_new = 4
    @grd_14 (railwayCrossing_a_new = FALSE ∧ railwayCrossing_a_new = railwayCrossing_b_new ∧
              railwayCrossing_b_new = railwayCrossing_c_new ∧ railwayCrossing_c_new = railwayCrossing_d_new) ∨
              (railwayCrossing_a_new = TRUE ∧ railwayCrossing_a_new = railwayCrossing_c_new) ∨
              (railwayCrossing_b_new = TRUE ∧ railwayCrossing_b_new = railwayCrossing_d_new) ⇒
                                                                    railwayCrossing_mode_new = 0
  then
    @act_0 railwayCrossing_mode ≔ railwayCrossing_mode_new
    @act_1 railwayCrossing_a_IO ≔ railwayCrossing_a_new
    @act_2 railwayCrossing_b_IO ≔ railwayCrossing_b_new
    @act_3 railwayCrossing_c_IO ≔ railwayCrossing_c_new
    @act_4 railwayCrossing_d_IO ≔ railwayCrossing_d_new
  end

  event RailwayCrossing_from_a_to_c
   where
    @grd_0 railwayCrossing_mode = 1
    @grd_1 railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE
   then
    @act_0 railwayCrossing_mode ≔ 0
    @act_1 railwayCrossing_c_IO ≔ railwayCrossing_a_IO
  end

  event RailwayCrossing_from_c_to_a
   where
    @grd_0 railwayCrossing_mode = 3
    @grd_1 railwayCrossing_b_IO = FALSE ∧ railwayCrossing_d_IO = FALSE
   then
    @act_0 railwayCrossing_mode ≔ 0
    @act_1 railwayCrossing_a_IO ≔ railwayCrossing_c_IO
  end

  event RailwayCrossing_from_b_to_d
   where
    @grd_0 railwayCrossing_mode = 2
    @grd_1 railwayCrossing_a_IO = FALSE ∧ railwayCrossing_c_IO = FALSE
   then
    @act_0 railwayCrossing_mode ≔ 0
    @act_1 railwayCrossing_d_IO ≔ railwayCrossing_b_IO
  end

  event RailwayCrossing_from_d_to_b
   where
    @grd_0 railwayCrossing_mode = 4
    @grd_1 railwayCrossing_a_IO = FALSE ∧ railwayCrossing_c_IO = FALSE
   then
    @act_0 railwayCrossing_mode ≔ 0
    @act_1 railwayCrossing_b_IO ≔ railwayCrossing_d_IO
  end
end
```

Appendix E

The complete model of the generic component

```
machine GenericComponent_Behaviour
variables GenericComponent_I GenericComponent_O GenericComponent_mode GenericComponent_IOrelation
invariants
 theorem @GenericComponent_thm0_0 ∀ps,s·ps ∈ ℙ1(ℤ) ∧ finite(ps) ∧ s ∈ ℙ(ps) ∧ card(s) = card(ps) ⇒ s = ps
 @GenericComponent_inv0_0 GenericComponent_I ∈ ℙ1(ℤ)
 @GenericComponent_inv0_1 GenericComponent_O ∈ ℙ1(ℤ)
 @GenericComponent_inv0_2 GenericComponent_mode ∈ 0..1
 @GenericComponent_inv0_3 GenericComponent_IOrelation ∈ GenericComponent_I ↔ GenericComponent_O
 @GenericComponent_inv0_10 finite(GenericComponent_I) ∧ finite(GenericComponent_O)
 @GenericComponent_inv0_11 dom(GenericComponent_IOrelation) = GenericComponent_I
 @GenericComponent_inv0_12 ran(GenericComponent_IOrelation) = GenericComponent_O
 @GenericComponent_inv0_13 GenericComponent_mode = 0 ⇒ GenericComponent_O =
                                                    GenericComponent_IOrelation[GenericComponent_I]
events
 event INITIALISATION
  then
   @GenericComponent_act0_0 GenericComponent_mode ≔ 0
   @GenericComponent_act0_1 GenericComponent_I, GenericComponent_O, GenericComponent_IOrelation :|
                GenericComponent_I' ∈ {i | i ∈ ℙ1(ℤ) ∧ finite(i)} ∧
                GenericComponent_O' ∈ {o | o ∈ ℙ1(ℤ) ∧ finite(o)} ∧
                GenericComponent_IOrelation' ∈ GenericComponent_I' ↔ GenericComponent_O' ∧
                dom(GenericComponent_IOrelation') = GenericComponent_I' ∧
                ran(GenericComponent_IOrelation') = GenericComponent_O'
 end

 // This event is needed to show that a component reads the inputs
 event GenericComponent_environment    where
   @grd0_0 GenericComponent_mode = 0
  then
   @act0_0 GenericComponent_mode ≔ 1
   @act0_1 GenericComponent_I ≔ dom(GenericComponent_IOrelation)
 end

 // This event illustrates the production of the outputs
 event GenericComponent_process_inputs    where
   @grd0_0 GenericComponent_mode = 1
  then
   @act0_0 GenericComponent_mode ≔ 0
   @act0_1 GenericComponent_O ≔ GenericComponent_IOrelation[GenericComponent_I]
 end
end
```

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

ISBN 978-952-12-3310-4
ISSN 1239-1891