Visual Component-based Development of Formal Models

Sergey Ostroumov, Marina Waldén
Faculty of Science and Engineering
Åbo Akademi University
Turku, Finland
E-Mail: {Sergey.Ostroumov, Marina.Walden}@abo.fi

***Abstract*—Formal methods, such as Event-B provide a means for system-level specification and verification supported by correctness proofs. However, the formal Event-B specification of a system requires background knowledge, which prevents a fruitful communication between the developer and the customer. In addition, scalability and reusability are limiting factors in using formal methods, such as Event-B in complex system development. This paper presents an approach to facilitate scalability of formal development in Event-B. Our aim is 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 that eases the rigorous development by applying the "drag-and-drop" approach and enhances the communication between a developer and a customer. We present a subset of components from the digital hydraulics domain and outline the compositionality mechanism.**

Keywords-Components Library; Visual Design; Event-B; Formal Components.

I. 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 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 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]. It 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 and reusability of this approach. Moreover, as more details are added to the specification through refinement, it is harder to convince the stake holders about the fact that the system specification embodies all the necessary requirements.

This paper proposes an approach to visual system design whose aim is to enhance scalability and reusability, as well as to facilitate the communication between a developer and a customer. In addition, the visual design is aimed at making the rigorous development process easier. The idea behind our approach is 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 twofold: a visual model whose correctness is supported by the underlying Event-B language. We present a pattern for the development of formal components and create a subset of components from the digital hydraulics domain. We also outline the compositionality mechanism.

The paper remainder is as follows. Section II outlines the Event-B notation and outlines proof obligations that provide the correctness proof. Section III presents the formal library of parameterized visual components. Section IV outlines the compositionality mechanism. Section V gives an overview of the existing approaches. Finally, Section VI concludes the paper and summarizes the directions of our future work.

II. PRELIMINARIES: EVENT-B

Event-B [1] is a state-based formalism that offers several advantages. First, it allows us to build system level models. Second, the development follows the top-down refinement approach, where each step is shown correct by mathematical proofs. Finally, it has a mature tool support extensible with plug-ins, namely the Rodin platform [3]. Currently, Event-B is limited to modelling discrete time, but the work on its extension to continuous models is on-going [4].

An Event-B specification consists of *contexts* and *machines*. A context can be *extended* by another context whereas a machine can be *refined* by another machine. Moreover, a machine can refer to the contents of the context via "*sees*" (see Figure 1).

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

 ISBN: 978-1-61208-553-1

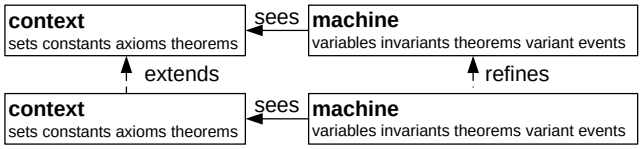


Figure 1. Event-B contexts, machines and relationship [1].

A machine models the behaviour of a system. The machine includes *state variables*, *theorems*, *invariants* and *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. 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, the system terminates or deadlocks. Sometimes, the system should never terminate, i.e., it has to be deadlock free. To achieve this, one needs to postulate a machine theorem that requires the disjunction of the guards of all the events to hold.

When a transition takes place, the action a is performed. The action a is a parallel composition ($\|$) of the assignments to the state variables executed simultaneously. An assignment can be either deterministic or non-deterministic. The 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 (w might include v). The non-deterministic assignment that we use in this paper is specified as $v :\in Q$, where Q is a set of possible values.

These denotations allow for describing semantics of Event-B in terms of *before-after predicates* (BA) [5]. Essentially, a transition is a BA that establishes a relationship between the model state before (v) and after (v') the execution of an event. This enables one to prove the model correctness by checking if the events preserve the invariants ($Inv \wedge g_E \Rightarrow [BA_E]Inv$) and are feasible to execute in case the event action is non-deterministic ($Inv \wedge g_E \Rightarrow \exists v' . BA_E$).

The refinement relation between the more abstract and more concrete specifications is also corroborated by the correctness proofs. Particularly, the more concrete events have to preserve the functionality of their abstract counter parts [6]. This paper however does not focus on this aspect.

The Rodin platform [3], tool support for Event-B, automatically generates and attempts to discharge (prove) the necessary proof obligations (POs). The best practices encompass the model development in such a manner that 90-95% of the POs are discharged automatically. Nonetheless, the tool sometimes requires user assistance provided via the interactive prover.

III. LIBRARY OF FORMAL COMPONENTS

Our idea is to create a formal library of visual 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. The system specification/development is then a process of picking, instantiating and connecting the needed components, so that the system is developed in the "drag-and-drop" fashion.

At present, the library contains components from the digital hydraulics and railway domains. The library also includes a generic component used to create a placeholder to be replaced by a specific one. Although our library consists of generic components parameterized for reuse, one can see that our approach is related to the work on domain specific languages, where the language is aimed at a specific problem domain [7][8]. Despite this, the formal language behind the components is Event-B and not domain specific.

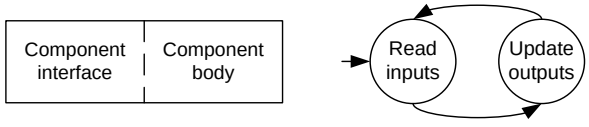
Next, we present a pattern for the component development and overview some components from the digital hydraulics domain, namely an electro-valve and a cylinder. We focus on the crucial parts of the models whose details, as well as more examples can be found in our TR [9].

A. Component Functionality

We start by describing the generic functionality of a component. A component is a reactive device that updates its outputs according to the input stimuli. The component typically consists of two parts: an interface and a body (Figure 2, 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.

The operation of the component has to be deterministic in order to precisely determine the output result. 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 [10] in which the component first reads the inputs (environment) and then produces the outputs (control). In other words, a component has at least two indefinitely alternating modes: read of the inputs and production of the outputs (Figure 2, b)). Thus, the non-termination (deadlock freedom) is the main property of a component.

We model components as Event-B machines that contain shared variables and rely on the principle of shared variables



a) b)

Figure 2. A component pattern: a) component structure, b) automaton.

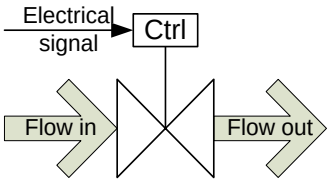


Figure 3. A symbolic representation of an electro-valve with the interface.

composition within Event-B when composing the components [11][12]. The variables that are local to a machine are considered private, while the shared variables are shared between machines and provide communication facilities in form of inputs and outputs. The inputs and the outputs of a component also form the interface of the component and are distinguished by the suffixes _I and _O (e.g., in Event-B we could have an input variable in_I and an output variable out_O).

B. *Hydraulic component: an electro-valve*

As an example of a parameterized visual component we develop and add to the library an electro-valve. Its visual symbol is shown in Figure 3 whereas the corresponding formal model is illustrated by Figure 4 and Figure 5.

The electro-valve is a physical device that transfers a flow of liquid from one port to another. It contains a plunger controlled by an electrical signal. The application of a positive control signal moves the plunger, so as to open the valve, whilst the negative signal closes it. If no signal is present on the control input, the plunger and therefore the valve keep the current position. Moreover, the valve opens and closes with some rate due to physical laws. The specification of a valve then has the following parameters (**context** Valve_parameters in Figure 4): the minimum (**valve_flow_min**) and the maximum (**valve_flow_max**) 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 (**valve_rate** ≤ **valve_flow_max** − **valve_flow_min**). Assuming that when the valve is closed, so that the outlet is fully closed as well (no flow can come through), the minimum flow equals to zero and the rate cannot be greater than the maximum. Moreover, if the rate equals to the maximum, the valve is simply open or closed. The minimum flow, the maximum flow and the rate parameters, as well as the set of control signals (**valve_CONTROL**) are all captured by constants in the context Valve_parameters (Figure 4).

The interface of a valve consists of two inputs and one output, namely the control signal (valve_control_I), the input port (valve_flow_I) and the output port (valve_flow_O), respectively (see Figure 5). Additionally, the valve has a variable that shows the current position of the plunger (valve_position), as well as the mode variable (valve_mode) that models the deterministic order of the transitions between the inputs read and outputs production states.

The valve has the property 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 always has to be 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 (see in Figure 5,

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

Figure 4. Parameters of a generic 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_flow_min..valve_flow_max ∧
  valve_flow_O ∈ valve_flow_min..valve_flow_max ∧
  valve_position ∈ valve_flow_min..valve_flow_max ∧
// The output flow cannot be stronger than allowed nor input
  valve_flow_O ≤ valve_position ∧
  (valve_mode = 0 ⇒ valve_flow_O ≤ valve_flow_I)
// The property of non-termination
theorem (valve_mode = 0 ∨
  (valve_mode = 1 ∧ valve_control_I = 1 ∧
    valve_position + valve_rate ≤ valve_flow_max) ∨
  (valve_mode = 1 ∧ valve_control_I = −1 ∧
    valve_position − valve_rate ≥ valve_flow_min) ∨
  (valve_mode = 1 ∧ (valve_control_I = 0 ∨
    (valve_control_I = 1 ∧
      valve_position + valve_rate > valve_flow_max) ∨
    (valve_control_I = −1 ∧
      valve_position − valve_rate < valve_flow_min))))
events  ...
 event valve_environment
  where  valve_mode = 0
  then  valve_mode := 1 || valve_control_I :∈ valve_CONTROL ||
    valve_flow_I :∈ valve_flow_min..valve_flow_max
 end

 event valve_opening
 any valve_flow_O_new
  where  valve_control_I = 1 ∧  valve_mode = 1 ∧
   (valve_position + valve_rate ≤ valve_flow_max) ∧
   (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
end
```

Figure 5. The excerpt of the machine of a generic valve.

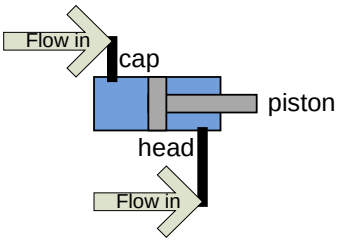


Figure 6. Visual representation of a cylinder.

theorem (valve_mode = 0 ∨ ...), which evaluates to true and supports the fact that the component always works.

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.

In order for a valve to produce the intended outputs, the valve first needs to read the inputs. This is captured by an environmental event that updates the inputs of the model. We assume that all inputs of the valve are updated simultaneously as shown in **event** valve_environment in Figure 5. The input flow is read non-deterministically bounded to the parameters of the valve.

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 events shown below.

The valve opening event (**event** valve_opening) 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 exceed the maximum (valve_position + **valve_rate** ≤ **valve_flow_max**). When the valve is opening, the output flow increases according to the rate 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 a flow 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).

The valve closing event is specified similarly considering the fact that it is opposite to the opening of the valve. It can take place when the command is to close the valve (valve_control_I = −1) and proceeds as long as the valve is not completely closed (valve_position − **valve_rate** ≥ **valve_flow_min**).

```
context Cylinder_parameters
constants
  cylinder_input_flow_min  cylinder_input_flow_max
  cylinder_cap_pos  cylinder_head_pos
axioms
  cylinder_input_flow_min = 0 ∧ cylinder_cap_pos = 0 ∧
  cylinder_input_flow_max ∈ ℕ1 ∧ cylinder_head_pos ∈ ℕ1
end
```

Figure 7. Parameters of a cylinder.

Finally, if the command is neither open nor closed (valve_control_I = 0) or the valve is fully closed or open, it keeps its position. 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) or the input flow (valve_flow_I < valve_flow_O ⇒ *valve_flow_O_new* = valve_flow_I).

The visual symbol and the specification of the electro-valve component extend the formal library of visual components. The specification was modelled and proved in the Rodin platform. The tool generated 24 POs out of which 20 were proved automatically.

C. *Hydraulic component: a cylinder*

Another example of a hydraulic component for the component library is a cylinder. The cylinder reacts on liquid flows only and does not have any electrical inputs. Nonetheless, it is a reactive device whose outputs are updated according to the input stimuli. The visual symbol of a cylinder is shown in Figure 6.

The cylinder contains a piston that can move forward and backward in the cylinder body depending on the differences between the liquid flows. The liquid flows via the cap and the head into the cylinder and is transformed into piston movement. The piston moves forward (extends) if the pressure of the flow coming into the cap is greater than the liquid flow coming into the head. In the opposite case, the piston moves backward. Clearly, if the pressure of both input flows is the same, the piston keeps the position. Due to physical laws, the piston moves with some rate. This rate is also determined by the difference in the input flows.

The cylinder specification has four parameters (Figure 7). Two of them define the minimum (**cylinder_input_flow_min**) and maximum (**cylinder_input_flow_max**) input flow of the liquid. We assume that both inputs are of the same size, so that the motion of the piston is proper. The other two parameters specify the limits of the piston motion (**cylinder_head_pos** and **cylinder_cap_pos**). The difference between **cylinder_head_pos** and **cylinder_cap_pos** sets the length that the piston can move.

The interface of the cylinder has two inputs (flows) (cylinder_flow_cap_I and cylinder_flow_head_I), as well as one output cylinder_piston_position_O (see Figure 8). The inputs allow the liquid to flow into the body of the cylinder via the cap and the head. The output of the cylinder is the piston that moves according to the difference in the input flows. Moreover, there is a variable that specifies the modes of the cylinder component, cylinder_mode (Figure 8). The main property of the cylinder is the deadlock freedom theorem. The theorem evaluates to true, which supports the fact that the cylinder is non-terminating.

Initially, there are no input flows, the piston is at some position within the cylinder body and the mode is set to read the inputs. In order for the piston to move, both of the inputs have to be updated (similar to the valve component).

There are three possible reactions to the input flows. The piston can move forward (extend), if the flow coming into

 ISBN: 978-1-61208-553-1

```
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_flow_min..cylinder_input_flow_max ∧
  // Input to move the piston to the left
  cylinder_flow_head_I ∈
      cylinder_input_flow_min..cylinder_input_flow_max ∧
  cylinder_mode ∈ 0..1 ∧
  // Deadlock freedom – non-termination
theorem cylinder_mode = 0 ∨
    (cylinder_mode = 1 ∧
     cylinder_flow_cap_I > cylinder_flow_head_I ∧
     cylinder_flow_cap_I > cylinder_input_flow_min ∧
     cylinder_piston_position_O + cylinder_flow_cap_I –
     cylinder_flow_head_I ≤ cylinder_head_pos) ∨
     ... // Guards of other events
```

Figure 8. Variables and properties of a cylinder.

the cap is larger than the flow coming into the head (cylinder_flow_cap_I > cylinder_flow_head_I). Moreover, the flow must be present on the cap input (cylinder_flow_cap_I > cylinder_input_flow_min) 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 equal to the difference between the input flows (Figure 9). The piston retracting is modelled in a corresponding manner.

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 (Figure 10.). The complete formal model of a cylinder can be found in [9].

```
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_flow_min ∧
    cylinder_piston_position_O + cylinder_rate ≤
                                        cylinder_head_pos
  then
     cylinder_mode := 0 || cylinder_piston_position_O :=
       cylinder_piston_position_O + cylinder_rate
  end
```

Figure 9. Forward motion of the piston (extend).

```
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
```

Figure 10. Keep the position of the piston (stop).

IV. RIGOROUS DESIGN USING THE LIBRARY

Once the components are developed and added to the library, one can (re)use/instantiate them while designing a system. The idea behind rigorous design with the library is the use of the "drag-and-drop" approach. Specifically, the developer picks and instantiates the necessary components by providing specific values for the parameters, a component name and adds them to the system model (Figure 11).

A. *Composition of decomposed machines*

The components can be seen as sub-unit machines which can be composed via parallel composition (||) [11][13]. For example, the machines A and B are composed into the (system) machine A || B, where the variables, invariants and events of A and B are merged. Overlapping variable and event names are renamed before composition. Note that composition is associative and commutative, but it cannot be reversed.

A way of refining a system is to superpose a new feature on its existing model (specification). The existing model is left unchanged while new variables and events modifying them are added to the model. The superposed feature and the existing model can be seen as components that can be composed. All these components in form of features or existing models are here considered to form library components. In addition, the composed models can form new library components.

The library components to be composed are connected via a connector. A connector is represented as a shared variable of a system machine whose mission is to promote the value of the output from one component to the input of the other one. Figure 12 illustrates a generic composition of two machines Component_n and Component_m into a single system machine System_M. The system model embodies the parameters of the components, their interfaces (environment events) and the connections between them. The functional events of the components are stored in separate machines and are *included* in the system.

B. *Composition of library components*

To show the connectivity mechanism, we will use a part of the Landing Gear (LG) case study whose details and formal model are described in [14]. Here, we will only show the connectivity of the valve and cylinder components as

 ISBN: 978-1-61208-553-1

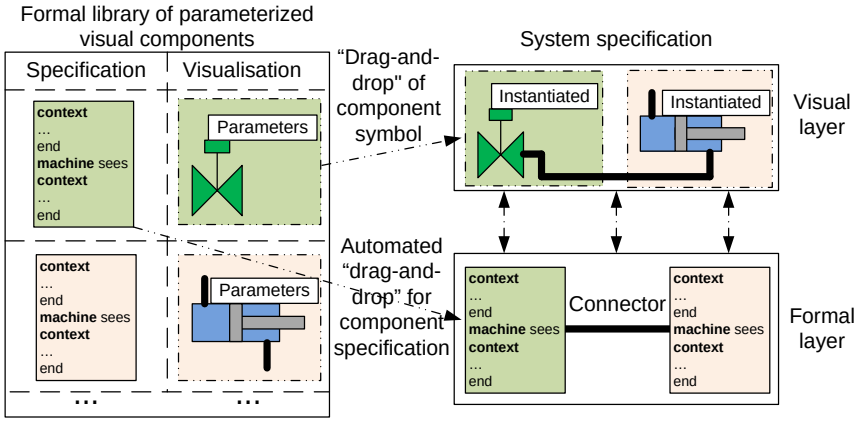


Figure 11. "Drag-and-drop" approach for visual system design in Event-B.

```
context SystemC
constants  SYSTEM_CONTROL
  // Component n constants
  // Component m constants
axioms  SYSTEM_CONTROL = {0,1,2}
  // Component n axioms
  // Component m axioms
end
```

```
machine System_M sees System_C
includes  Component_n  Component_m
variables  Control  connection_Comp_n_Comp_m
  // Shared variables of Component n
  // Shared variables of Component m
invariants  Control ∈ SYSTEM_CONTROL ∧
  connection_Comp_n_Comp_m ∈ <COMPONENT_n_OUTPUT_TYPE>
  // Component n invariants
  // Component m invariants
variant  max(SYSTEM_CONTROL) – Control
events
  event INITIALISATION extends INITIALISATION then
     Control := 0 || connection_Comp_n_Comp_m := <INIT_VALUE>
  end

  event Comp_n_environment refines Comp_n_environment
    where  ... // Guards derived from component n
      ∧  Control = 0
    then  ... // Actions derived from component n
      || Control := 1
  end

  convergent event system_connection_Component_i_Component_k
    where  Control = 1 ∧ <Component_n_mode> = 0
      // Ensure that the component n has updated its outputs
    then  Control := 2 ||
      connection_Comp_n_Comp_m:= <Comp_n_Out>
  end

  event Comp_m_environment refines Comp_m_environment
    where  ... // Guards derived from the component m
      ∧ Control = 2
    then  ... // Actions derived from the component m
      || Control := 0
  end
end
```

Figure 12. Composition of Component n and Component m machines.

visually depicted in Figure 11. More details about various components, connectivity mechanisms and refinement patterns, can be found in the technical reports [9][14].

The main purpose of the LG system is to extend the landing wheels (connected to the hydraulic cylinders) when an airplane is to be landed and to retract them during the flight. The extension/retraction of the cylinders is controlled by the valves. Thus, the valves are connected to the cylinders sequentially (see Figure 11, visual layer).

The formal layer of the visual representation of Figure 11 is shown in Figure 13 and Figure 14. The context machine contains the constants and axioms of the valve and the cylinder. The theorem supports the connectivity between the components. It shows that the output of the source component is compatible with the input of the target component. Generally, the maximum diameter of the valve output should be the same as the maximum input flow of the cylinder connected to it.

The system machine LG_System_M includes the library components valve (Valve_Behaviour) and cylinder (Cylinder_Behaviour) (see Figure 14). The connectivity between these components is represented by the variable connection_Valve_Cylinder_head. When the valve updates its

```
context LG_System_C
constants  CONTROL_HEAD
  valve_0_flow_min  valve_0_flow_max  valve_0_CONTROL
  valve_0_rate  cylinder_0_cap_pos  cylinder_0_input_flow_min
  cylinder_0_input_flow_max  cylinder_0_head_pos
axioms
  // valve_0
  valve_0_flow_min = 0 ∧  valve_0_flow_max = 10 ∧
  valve_0_CONTROL = {−1,0,1} ∧ valve_0_rate = valve_0_flow_max ∧
  // cylinder_0
  cylinder_0_input_flow_min = 0 ∧ cylinder_0_input_flow_max=10 ∧
  cylinder_0_cap_pos = 0 ∧ cylinder_0_head_pos ∈ ℕ1 ∧
   // system_1
   CONTROL_HEAD = {0,1,2}
  theorem // system_1
    cylinder_0_input_flow_max = valve_0_flow_max
end
```

Figure 13. The parameters of the LG system: a valve, a cylinder and system parameters.

 ISBN: 978-1-61208-553-1

output value (i.e., when its mode is 0), this value is then used to update the value of the connector (connection_valve_cylinder_head := valve_0_flow_O in convergent event Connection_Valve_Cylinder). This value is in turn used as the input to the cylinder (cylinder_0_flow_head_I := connection_valve_cylinder_head in event cylinder_0_environment). Hence, the overall scheme is as follows. First, the valve inputs are updated, so that the valve component can update its output. Then, the value of the connector is updated according to the valve output. Finally, the inputs of the cylinder are updated according to the value of the connector.

Several connectors can be added in one refinement step following the same pattern. The proof of the connectivity mechanism relies on the superposition refinement rule, where the machine of the composed system refines the machine of each component.

```
machine LG_System_M sees LG_System_C
includes Valve_Behaviour Cylinder_Behaviour
variables Control_head connection_valve_cylinder_head
 valve_0_control_I valve_0_flow_I valve_0_flow_O
 valve_0_mode valve_0_position
 cylinder_0_piston_position_O cylinder_0_flow_cap_I
  cylinder_0_flow_head_I cylinder_0_mode
invariants
 ... // Valve_0 type definitions and main invariants
 ... // Cylinder_0 type definitions and main invariants
 control_head ∈ CONTROL_HEAD ∧
 connection_Valve_Cylinder_head ∈
  cylinder_0_input_flow_min .. cylinder_0_input_flow_max
variant max(CONTROL_HEAD) - control_head

events
 ...
 event valve_0_environment refines valve_0_environment
  where
   mode = 0 ∧ control_head = 0
  then
   valve_0_mode := 1 || valve_0_control_I :∈ valve_0_CONTROL ||
   valve_0_flow_I := <INPUT> || control_head := 1
 end

 convergent event Connection_Valve_Cylinder
  where
   valve_0_mode = 0 ∧ control_head = 1
  then
   control_head := 2 ||
   connection_valve_cylinder_head := valve_0_flow_O
 end

 event cylinder_0_environment
  where
   cylinder_0_mode = 0 ∧ control_head = 2
  then
   cylinder_0_mode := 1 || cylinder_0_flow_cap_I := <NEW_VALUE>
  || cylinder_0_flow_head_I := connection_valve_cylinder_head
  || control_head := 2
 end
end
```

Figure 14. An instantiated valve connected with an instantiated cylinder.

V. Related Work

BMotionStudio has been proposed as an approach to visual simulation of the Event-B models [15][16]. The idea behind BMotionStudio is that the designer creates a domain specific image and links it to the model using a "gluing" code written in JavaScript. The simulation is based on the ProB animator and model checker [17], so that whenever the model is executed the corresponding graphical element reacts is updated. The BMotionStudio tool also supports interaction with a user – the user can provide an input via visual elements instead of manipulating the model directly.

In contrast to the BMotionStudio approach, we aim for creating visual descriptions of models via a library of predefined components that have a formal, as well as a 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 of the components, but simply to reuse them. Eventually, the designer obtains a graphical representation of the system whereas its specification is in fact written in Event-B and supported by correctness proofs. Certainly, our approach can be complemented by BMotionStudio in order to obtain visualisation of the model execution.

Snook and Butler [18] proposed an approach to merge visual UML [19] with B [20]. The latter is supposed to give a formal precise semantics to the former at the same time as the former is aimed at reducing the effort in training to overcome the mathematical barrier. This approach has then been extended to Event-B and is called iUML-B [21]. The authors define semantics of UML by translating it to Event-B. The use of the UML-B profile provides specialisation of UML entities to support refinement. The authors also present tools that generate an Event-B model from UML.

A component based reuse methodology for Event-B was presented by Edmunds et al. [22], where the composition is based on the shared events principle. Their idea is to have a library of Event-B components where the component instances and the relationships between them are represented diagrammatically using an approach based on iUML-B.

Instead of using UML as a visualisation tool as in both the above cases, 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 [23]. The aim of this work is the generation of a structural VHDL [24] 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 VHDL library components. The authors rely on an additional refinement step where regular operations are replaced with function calls. This allows for automated generation of structural VHDL descriptions.

 ISBN: 978-1-61208-553-1

Instead of focusing on code generation, 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. Our goal is to create a formal library of parameterized Event-B specifications that capture the generic behaviour of these components. Our approach is to facilitate component reuse, where the developers can specify systems in a "drag-and-drop" manner.

VI. Conclusion and Future Work

We have proposed an approach to the development of rigorous components augmented with unique graphical symbols. It is based on the pattern that allows seamless integration of components into a system. We have illustrated the proposed approach using components from the digital hydraulics domain, where each component has been formally developed and proved correct within Event-B. The components constitute the library, which captures the graphical representations, formal specifications and a one-to-one relation between them. The library enables components reuse and instantiation in various applications depending on the requirements. In addition, visual design structures the specifications and facilitates scalability of the rigorous development. Moreover, it is useful in the communication between developer and customer. This will need an evaluation via empirical studies comparing our approach to the traditional formal development. We believe that the proposed approach is applicable to other than Event-B formalisms as well considering their syntactical specifics.

The components connectivity outlined in this paper is an important element of systems development. We are currently extending this mechanism considering various types of connections and stepwise refinement. Moreover, the tool support is one of the key factors for facilitating an easy access to the proposed approach. Thus, our future work also includes providing the tool support, which will include an interface to "drag-and-drop" components, maintenance and extension of the library, as well as automated application of the connectivity patterns through instantiation in order to derive a composed system. The proofs will be conducted via the tool support for Event-B.

Acknowledgment

The authors would like to thank Dr. Marta Olszewska and Dr. Andrew Edmunds for the fruitful discussions. The work was done within the project ADVICeS funded by the Academy of Finland, grant No. 266373.

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, New York: Springer-Verlag, 1998.

[3] RODIN IDE. [Online]. Available from: http://sourceforge.net/projects/rodin-b-sharp/, February 2017.

[4] R. Banacha, M. Butler, S. Qinc, N. Vermad, and H. Zhue, "Core Hybrid Event-B I: Single Hybrid Event-B machines", Science of Computer Programming, vol. 105, Elsivier, pp. 92-123, 2015.

[5] C. Métayer, J.-R. Abrial, and L. Voisin, Event B language, vol. 3.2, RODIN Deliverables. [Online]. Available from: http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, May 2005.

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

[7] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography", vol. 35(6), SIGPLAN Notices, pp. 26–36, 2000.

[8] P. Boström, Formal Verification and Design of Systems using Domain Specific Languages, TUCS Dissertations 110, 2008.

[9] S. Ostroumov and M. Waldén, Formal Library of Visual Components, TUCS TR, vol. 1147. [Online]. Available: http://tucs.fi/publications/view/?pub_id=tOsWa15a, May 2015.

[10] M. Butler, E. Sekerinski, and 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.

[11] J.-R. Abrial, Event Model Decomposition, ETH Zurich TR, vol. 626. [Online]. Available from: http://wiki.event-b.org/images/Event_Model_Decomposition-1.3.pdf, April 2009.

[12] T. S. Hoang, A. Iliasov, R. A. Silva, and W. Wei, "A Survey on Event-B Decomposition", Workshop on Automated Verification of Critical Systems, vol. 46, Electonic Communication of the EASST, pp. 1-15, 2011.

[13] R. J. Back, "Refinement calculus, part II: Parallel and reactive programs", Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, vol. 430, LNCS: Springer-Verlag, pp. 67–93, 1990.

[14] S. Ostroumov and M. Waldén, Facilitating Formal Event-B Development by Visual Component-based Design, TUCS TR, vol. 1148. [Online]. Available from: http://tucs.fi/publications/view/?pub_id=tOsWa15b, September 2015.

[15] L. Ladenberger, J. Bendisposto, and M. Leuschel, "Visualising Event-B Models with B-Motion Studio", Workshop on Formal Methods for Industrial Critical Systems, vol. 5825, LNCS: Springer-Verlag, pp. 202-204, 2009.

[16] BMotion Studio for ProB Handbook. [Online]. Available from: https://www3.hhu.de/stups/handbook/bmotion/current/html/index.html, April 2015.

[17] M. Leuschel and M. Butler, "ProB: A Model Checker for B", Symposium of Formal Methods Europe, vol. 2805, LNCS: Springer-Verlag, pp. 855-874, 2003.

[18] C. Snook and 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.

[19] G. Booch, I. Jacobson, and J. Rumbaugh, Unified modeling language Reference Manual, The (2nd edition), USA: Pearson Higher Education, 2004.

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

[21] C. Snook and M. Butler, "UML-B and Event-B: an integration of languages and tools", IASTED Conference on Software Engineering, pp. 12-17, 2008.

[22] A. Edmunds, C. Snook, and M. Walden, "On Component-Based Reuse for Event-B", ABZ Conference on ASM, Alloy, B, TLA, VDM, and Z, vol. 9675, LNCS: Springer-Verlag, pp. 151-166, 2016.

[23] S. Ostroumov, L. Tsiopoulos, J. Plosila, and K. Sere, "Generation of Structural VHDL Code with Library Components From Formal Event-B Models", DSD Euromicro Conference, IEEE, pp. 111-118, 2013.

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

 ISBN: 978-1-61208-553-1